1991

The integration system for the Low Cost Combat Direction System

Bolick, Willie Kelly; Irwin, Richard Thomas

Monterey, California. Naval Postgraduate School

http://hdl.handle.net/10945/43778

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

THE INTEGRATION SYSTEM
FOR
THE LOW COST COMBAT DIRECTION SYSTEM

by

Willie Kelly Bolick
and
Richard Thomas Irwin

September 1991

Thesis Advisor: Dr. Valdis Berzins

Approved for public release; distribution is unlimited.

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Computer Science Dept. Naval Postgraduate School; 6b. OFFICE SYMBOL *(if applicable)* CS	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS *(City, State, and ZIP Code)* Monterey, CA 93943-5000	7b. ADDRESS *(City, State, and ZIP Code)* Monterey, CA 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL *(if applicable)*	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS *(City, State, and ZIP Code)*	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
THE INTEGRATION SYSTEM FOR THE LOW COST COMBAT DIRECTION SYSTEM

12. PERSONAL AUTHOR(S)
Bolick, Willie Kelly; Irwin, Richard Thomas

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
Master's Thesis	FROM 09/90 TO 09/91	1991, September, 10	278

16. SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
Combat Direction Center, Software Engineering, Integration System

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

In a world where changes in technology occur each minute, the demand for a hard Real Time embedded computer system deployed on board naval ships not equipped with Naval Tactical Data System increases. As the demand increases, an important fact looms, a new approach to software development and system design is essential. The approach used in our research started with the requirement specifying use of Ada as the design language with UNIX as the operating system, and selection of the commercial workstation rugged enough to withstand shipboard requirements. The system requires standard power with no special interface equipment for adaptation to shipboard application. Specific benefits include ease of maintenance and expansion of ongoing processes and applications, allowing the system to grow as the need grows.

This study provides a detailed set of requirements, functional specifications, designs, and a prototype implementation of the Integration System for such a system. The approach taken is to implement the basic features of a Combat Direction System (CDS) on a commercially available microprocessor workstation. This Integration System for the Low Cost Combat Direction System meets all the requirements specified by the Naval Sea Systems Command. The code provides the basic elements and is designed for integration of a database, a user interface, and the ships sensors necessary to provide essential data to operate the system.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [X] UNCLASSIFIED/UNLIMITED [ ] SAME AS RPT. [ ] DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Valdis Berzins	22b. TELEPHONE *(Include Area Code)* (408) 646-2461; 22c. OFFICE SYMBOL CS/Be

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Approved for public release; distribution is unlimited

THE INTEGRATION SYSTEM
FOR
THE LOW COST COMBAT DIRECTION SYSTEM

by

Willie Kelly Bolick
Lieutenant, United States Navy
B.S., University of Arkansas, 1977
M.S., Arkansas State University, 1980
and
Richard Thomas Irwin
Lieutenant, United States Navy
B.S., University of Michigan-Flint, 1983

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 1991

ABSTRACT

In a world where changes in technology occur each minute, the demand for a hard Real Time embedded computer system deployed on board naval ships not equipped with Naval Tactical Data System increases. As the demand increases, an important fact looms, a new approach to software development and system design is essential. The approach used in our research started with the requirement specifying use of Ada as the design language with UNIX as the operating system, and selection of the commercial workstation rugged enough to withstand shipboard requirements. The system requires standard power with no special interface equipment for adaptation to shipboard application. Specific benefits include ease of maintenance and expansion of ongoing processes and applications, allowing the system to grow as the need grows.

This study provides a detailed set of requirements, functional specifications, designs, and a prototype implementation of the Integration System for such a system. The approach taken is to implement the basic features of a Combat Direction System (CDS) on a commercially available microprocessor workstation. This Integration System for the Low Cost Combat Direction System meets all the requirements specified by the Naval Sea Systems Command. The code provides the basic elements and is designed for integration of a database, a user interface, and the ships sensors necessary to provide essential data to operate the system.

THESIS DISCLAIMER

Appropriate credit is given for names used which are trademarks of various corporations.

ADA is a registered trademark of the United States Government, ADA Joint Program Office.

LMS 11 is a registered trademark of LOGICON.

SUN is a registered trademark of Sun Microsystems.

UNIX is a registered trademark of AT&T.

The views expressed in this thesis are those of the authors and do not reflect the official policy or position of the Department of Defense or the United States Government.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

We owe a great debt of gratitude to many people who have inspired and encouraged us during the process of writing this thesis. Special thanks goes to our families, for their support and patience. Also, we would like to mention a few individuals for the time, energy, and advice they gave on behalf of our efforts to formulate our concepts, research, and code the final product:

Mr. Walter Landaker (NPS, Monterey)

Mr. John Locke (NPS, Monterey)

Mr. Russell H. Whalen (NPS, Monterey)

Mr. Mike Williams (NPS, Monterey)

Mr. Albert Wong (NPS, Monterey)

I. INTRODUCTION

The primary goal of the combat direction center is to ensure the individual fighting capabilities of a single ship. Each ship, however, not only supports the task force, but enhances it to make the task force a single fighting element capable of overcoming any enemy. The Navy has met the challenge of the 1990's with the development and implementation of the AEGIS System. Combatants without this AEGIS capabilities are being upgraded to meet these standards and capabilities when and where it is possible. In some cases this is impossible, for instance, most non-combatants at present have no automated capabilities whatsoever. The Navy had the choice of either starting from scratch and fitting these ships from ground up or developing a new system that was capable of meeting specific requirements, still holding the cost of development and implementation to an affordable level. The Navy has projected its desire to develop a system that can be installed on non combatant ships or to augment existing systems on Combat Direction System(CDS) equipment ships. This implementation would be accomplished in Ada and would reflect the specifics of five increments as detailed in Reference 1. The introduction of the Low Cost Combat Directions System (LCCDS) [Ref. 1, 2] into the field of research and development launched the need for a new look at the way Combat Direction Systems function.

The increased complexity of warfare in this decade and the next requires a system capable of timely response and rapid recovery. The LCCDS, a Real Time System, will meet this challenge. Receiving data from a number of sensors, the system will process raw data into formatted information which is both displayed and stored in the database for future recovery and use. Utilizing the Global Positioning System (GPS) the LCCDS will continuously monitor and update ownship position. Receive only link 11 provides a tactical picture of the ship's environment. Equally significant is the user interface, which provides

a variety of inputs from the operator and creating a well balanced, functional, and informative system capable of handling the most critical situation.

The LCCDS system will be implemented on a commercially available microprocessor-based workstation. Selection of a microprocessor is relatively straight forward.

1. The system must meet the NAVSEA requirements for shipboard use.

2. It must be capable of handling our software requirements.

The Sun Microsystems SPARCstation 2 will provide the capabilities required for the shipboard and real time environment of the LCCDS. The 4.2 BSD UNIX operating system has been suggested [Ref. 3] and meets the requirements necessary to manage the Verdix Ada software development system. Verdix Ada will be the implementation language for the Low Cost Combat Direction Software System. The integration must accomplish an interface between existing shipboard navigation sensors, link 11, and the object oriented database management system. These interface points with navigation and link 11 are not interactive, and allow the integration system only to receive data. The user interface will receive data from the integration system while the database will support both retrieval and storage of data via the integration system.

The LCCDS will accomplish all these tasks plus several additional services in just seconds vice minutes and with a much greater accuracy and reliability than manual methods. This capability is made possible by the careful selection of a powerful, inexpensive microprocessor workstation. One of the projected users of the LCCDS is on board ships without Naval Tactical Data System(NTDS), where at present handling of combat support is accomplished manually, using only maneuvering board and status boards kept updated by individual watch standers. The addition of the LCCDS to one of these platforms would leave the Commanding Officer and his watch standers free to accomplish their mission in a more accurate, safe, and expedient manner.

The integration system is a vital element(module) of the Low Cost Combat Direction System(LCCDS) project which is sponsored by Naval Sea Systems Command(NAVSEA). The LCCDS project is currently divided into three major research and development areas.

1. The **integration system**, whose primary function is confining and filtering information from several sources, including ownship sensors, Global Positioning System and receive only link 11. To monitor this information and detect impending significant events, such as closest point approach of other vessels, shoals, aircraft fly over, and navigation hazards. To provide, to the user, a means of continuous access to necessary navigation data, such as ownship fix information, position of intended movement, and waypoint locations. To provide an archival record of the available tactical information for both immediate and historical use.

2. The **user interface** module, which provides the user with onscreen visual elements to provide tactical information in an effective form and enables the user to manage the LCCDS. The user interface receives track information, and environmental information requested from the integration system.

3. The **navigation system** of the LCCDS, which will provide ownship navigation and maneuvering data.

The objective of this thesis is to describe the research and development of the integration system for the LCCDS. In conjunction with the development, a design and implementation phase for the integration system as a part of the LCCDS is discussed. A prototype of the integration system with full details on integrating the user interface, the navigation system, and an object oriented database is implemented in Ada. The integration system meets all the requirements of a real time systems as required in the design specifications [Ref. 1].

A. HISTORICAL BACKGROUND OF THE LCCDS

The traditional or conceptual meaning of a ship's combat system is typically the men and equipment which provide the ship with its offense and defense capabilities. However, some subsystems such as communication and navigation are not in the spotlight as often as the weapons system. Both subsystems accomplish their mission in a routine manner and unless disabled or inoperative are forgotten or de-emphasized when combat systems are discussed. These systems, which provide the eyes and ears for the ship, play an equally important role in the ship's overall combat system. It is the composite of the ship's elements and personnel processing either manual or automated information and providing support to the overall task/mission of the platform that is important. During the late 1950's and since the Naval Tactical Data System (NTDS) has played the role of tactical data integration. Since its evolution out of a need for faster and more accurate information NTDS has been plagued with restrictions and hang-ups. As technology increased, the need to improve the system increased, yet many of the outdated systems were not replaced, and heavy requirements for manual intervention and control continued to slow and restrict the system. Uncoordinated changes in the interfacing system and weapons systems cause a make shift and continuous catch up mode.

Today we have several different generations of these modified/improved systems in the fleet [Ref. 4]. Ongoing study and thirty years of experience has caused the development and deployment of the Combat Direction System which is not totally separated from, but has substantial increases in capabilities over the NTDS. The role of the Combat Direction System is composed as follows [Ref. 5].

1. An automated Database Management System capable of managing tactically significant tracks.

2. A combination of necessary element to form a combat system whose primary purpose is to support the combat direction center.

B. PROJECT ORGANIZATION AND GOALS

The Low Cost Combat Direction System research and development is under the supervision of the Naval Sea Systems Command. Research is ongoing at the Naval Post Graduate School, Monterey, CA., Naval Ocean Systems Command, San Diego, CA., and Massachusetts Institute of Technology, Cambridge, MA.

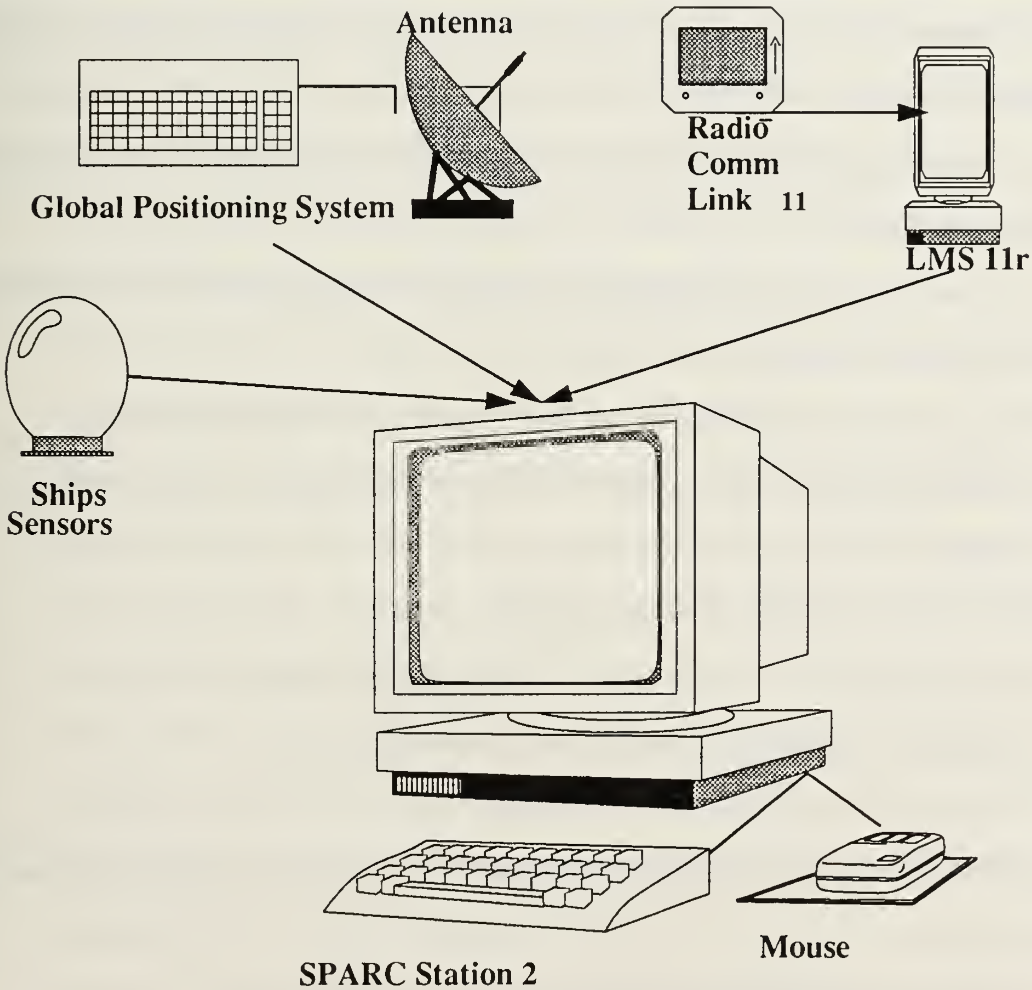


Figure 1 : LCCDS CONFIGURATION DIAGRAM

The LCCDS project as shown in Figure 1, is divided into three basic areas of development:

1. The hardware evaluation and procurement.

2. The development of the software packages.

3. Testing and evaluation for Real Time performance

The integration system Project, an element in areas 2 and 3 of the LCCDS, is divided into four major areas of research and development.

1. A system capable of providing an interface with the user, developing an interactive communication between user and system.

2. A system capable of interfacing with the navigation system and providing ownship navigation data.

3. A system capable of interfacing with Link 11 receive only and providing for display network track data.

4. A system capable of data storage and retrieval utilizing data received from sensor interfaces, and direct input from the user.

The project sponsor goals for the LCCDS integration system are as follows:

1. Locate, evaluate, and procure the hardware necessary to meet the shipboard requirements.

2. Use Ada as the implementation language.

3. Integrate an object-oriented Database Management System.

4. Integrate a manual tracking and identification capability.

5. Integrate a receive only link 11 capability.

6. Integrate an on ship navigation and maneuvering capability, along with display of shore line maps.

7. Integrate an autotracking capability [Enclosure 1, Ref. 1]

8. Test, evaluate, and employ the system.

The detailed initial problem statement can be defined in terms of a high level LCCDS program description. Develop the prototype of a Low Cost Combat Direction SoftWare System (LCCDSWS) for a Low Cost Combat Direction System (LCCDS) that implements the basic features of Combat Direction System "Model 5"on a commercially available microprocessor based workstation [Ref. 1,2,3]. This is to be accomplished in respect to the five increments as outlined in Enclosure 1 Reference 1. Based on these guidelines this phase of the research must then start at the beginning, laying into place each part of the puzzle, with a focus on ensuring that no piece will place a constraint on any other piece. In fact, our goal is that each piece will enhance all the remaining pieces. To start, we had to select a system and the software environment for the system. The next steps are to define the requirements for the integration system, write the functional specifications linking the user interface and navigation modules, then implement the above in Ada.

In order for the integration system to meet these requirements, specific goal definitions for the integration system have been established.

Goal 1. The integration system must provide a track database system, which is capable of accessing and updating track information in Real_Time.

Goal 2. The integration system must be able to parse incoming Global Positioning System(GPS) data and extract track/ownship location data in Real_Time.

Goal 3. The integration system must be able to parse incoming link 11 messages and extract track data in Real_Time.

Goal 4. The integration system must be able to parse incoming sensor related messages and extract track data in Real_Time.

Goal 5. The integration system must be able to provide the user with relevant tactical data external to the platform, for screen display.

Goal 6. The integration system must be able to provide the user with the ability to customize and organize data to meet the specific needs of the individual platform.

Goal 7. The integration system must be able to provide the user with the ability to limit the number of tracks and/or elements for display. Any or all of this data must be available for retrieval and display. The user will by means of a filter package communicate to the integration system what is to be displayed.

Goal 8. The integration system must be able to provide the user with the ability to store, manage, and display geographical regions, paths, and waypoints to meet the specific needs of the user.

Goal 9. The integration system must be able to provide the user with ownship data to include closest point of approach(CPA) time, bearing, and range. CPA data provided may be between any track and ownship or between any two tracks, and must be in Real_Time.

LT Bolick focused on requirements analysis, system specifications and the overall system design constraints. LT Irwin concentrated on the development of the software components. Both contributed to the system architectural analysis, software development, implementation and design.

C. SOFTWARE ENGINEERING APPROACH.

The software development process has been defined by several different and capable authorities as having different and varied meanings. Yet all seen to agree on some specific points. The first and most overwhelming point is that when starting a project, the specific requirements must first be defined, researched and redefined. The second point is, that a set of specifications must be developed and a design architecture presented before proceeding with development of the project. Following these well established guidelines [Ref. 7]the model for the LCCDS integration system was developed.

1. Requirements analysis [Ref. 3].
2. Functional specifications.

3. Architectural design.

4. Implementation.

5. Testing and Evaluation.

The first state in the LCCDS design, the requirements analysis, has been accomplished by the team of Seveney and Steinberg [Ref. 3]. It is our intention, however, to refine these broad requirements to more specific ones directly related to the integration system. At this point we focus on the initial problem statement: The thrust of this research is to provide detailed requirement analysis for the software portion of the LCCDS. We refer to this as the Low Cost Combat Direction Software System (LCCDSWS).

The Department of Defense(DOD) and Navy have taken great care in the development of specific guidelines for the design and implementation of software to be used by DOD. Directives to be considered in the integration system software require effort to be placed in:

1. Accomplishing the task (completion of the integration system).

2. Completion in a timely manner.

3. Completion at no significant additional cost to sponsor.

4. Producing a top quality product.

Using the spiral model of software development the following sequence of events have been established for the LCCDS integration system design, review, and acceptance.

1. Review and evaluation of requirements specified by the sponsor(NAVSEA).

2. Review and evaluation of requirements document (Masters Thesis by Seveney and Steinberg) to determine if there exist conflicts with the NAVSEA requirements.

3. Requirements Analysis Review(RAR) and consistent needs identified.

4. Needs analysis and new needs identified.

5. Completion of specifications with a review and evaluation of requirements and any new needs are identified.

6. Functionality review for first design.

7. Design review and reevaluation of needs and requirements. If necessary, apply changes to design.

8. Design accomplished with testing in progress. Review for requirements and needs by sponsor. Changes due to requirements and needs identified are applied at this time. Bugs are removed from software. Complete code review and code documentation. Module testing accomplished.

9. Design complete and ongoing testing and evaluation standards. Implementation of a working prototype. Complete system testing with independent quality assurance verification.

10. Delivery to sponsor, and ongoing maintenance and upgrade. (debugging in progress).

Research and design of the integration system conforms with the following DOD and Navy directives.

1. Department of Defense Military Standard 2167-A Defense System Software Development [Ref. 25].

2. Department of Defense Military Standard 2168 Defense System Software Quality Program [Ref. 26].

3. American National Standard Institute Military Standard 1815A-1983 Reference Manual for the Ada Programming Language [Ref. 27].

4. DOD-STD_480, Configuration Control_Engineering changes, Deviations, and waiver [Ref. 28].

5. MIL-STD-483,

6. MIL-STD-490, Specification Practices [Ref. 30].

7. MIL-STD-1388, Logistic Support Analysis

The following Data Item Description(DID):

1. DI-MCCR-80012, Software Design Document

2. DI-MCCR-80014, Software Test Plan

3. DI-MCCR-80017, Software Test Report

4. DI-MCCR-80025A, Software Requirements Specification

5. DI-MCCR-80026, Interface Requirements Specification

Data Item Description, DI-MCCR-80025A, Software Requirements Specification, specifies the engineering and qualification requirements for a computer software configuration item (CSCI). As the basis for the design, format, data generation, and formal testing of this software project, our team of designers, used the Software Requirements Specification noted above.

II. INTEGRATION SYSTEM RESEARCH ANALYSIS

A. REQUIREMENTS FOR LCCDS.

The initial problem statement can best be stated by paraphrasing the Enclosure 1 to Reference 1, "**Statement of work for Low Cost Combat Direction System (LCCDS)**" which outlines the five increments that the LCCDS project is to be divided.

In increment one:

1. A computer system is to be selected
2. Design and develop an object-oriented Database Management System.
3. Design and develop a display/graphics, which provides the user with his own customized screen format allowing interactive operations with the system.
4. Display tracks and ownership data.
5. General response time to user "should be no greater that one half second".

In increment two:

1. Integrate manual tracking and track identification capability.
2. System maintains ownership track.
3. Use standard display symbols as list in Reference 1.
4. Display and assign speed and bearing as both values and leaders, with four second updates on all elements of the each track in the database.
5. Allow for additional/amplifying track information to be displayed at the users request.
6. Allow the user to change track identification number, category, and identity.
7. Allow for a unlimited number of tracks in the system.

In increment three:

1. Integrate receive only link 11.

In increment four:

1. Provide ownship data. Navigation and maneuvering data from ownership sensors.

2. Provide up to six steaming routes.

3. Provide up to 50 waypoints per steaming routes.

4. Provide closest point approach data.

a. Provide ownship CPA with any track.

b. Provide CPA between any two tracks.

c. Provide display of CPA bearing lines on position display.

In increment five:

1. Integrate an organic auto tracking capability using (TBD) radar interface.

Issues in achieving common operations for Combat Direction Systems was addressed in accordance with the guidelines of Reference 2. The specific concerns faced by this research study and the issue we considered most important is safeguarding consistency, while preserving independent configurations for each user. A list of considerations by which to achieve this concerns are listed below.

1. What track characteristics should be specified in statements. Should the track follow the basic NTDS format.

2. What actions should the system take in the event of malfunction or error detected and what actions are left to the user.

3. Which of the common display and control formats of the model 5 Combat Direction System should be used.

4. What safeguards should be built into the system, more specifically the integration system, to insure consistent operations.

5. What accuracy and precision of track data is required.

Communications between the integration system and the elements of the LCCDS is a critical link in considering development of a Real_Time system. There cannot be any delay in the system functions due to restrictions in the communications media. Therefore care and time was used in the selection and implementation of the communication software interface between the three elements user interface, Link 11, navigation interface, and the integration system as seen in Figure 2.

It is important to keep these requirements in mind, not allowing them to drive the research, but to provide some guidelines and restrictive boundaries within which to work.These questions and more are addressed and answered in this research.

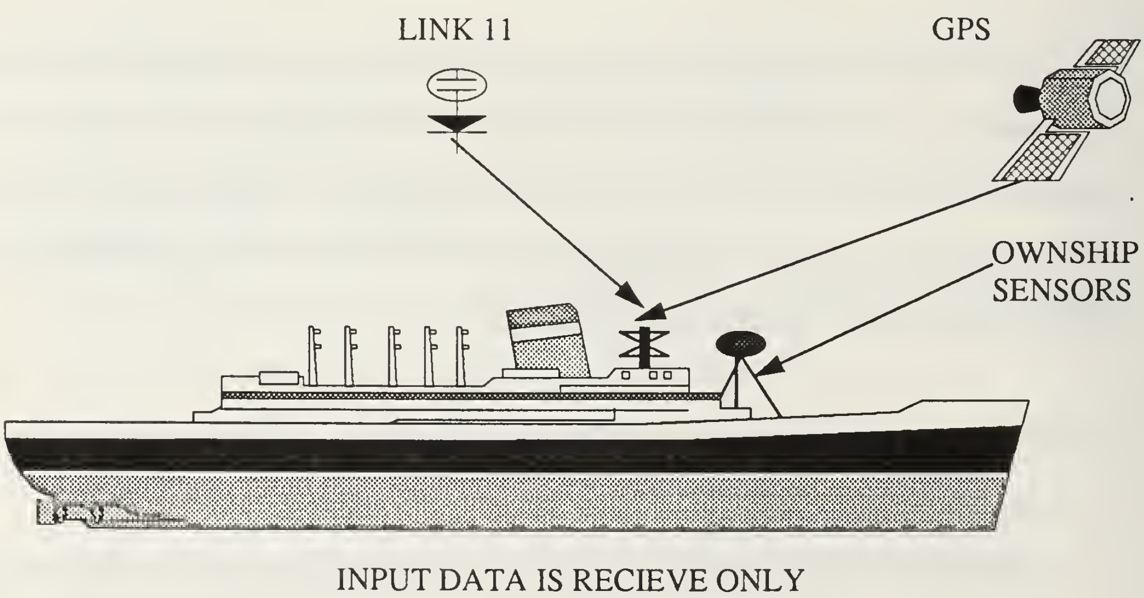


Figure 2 : NON-NTDS PLATFORM

B. LOW COST COMBAT DIRECTION SYSTEM CONTEXT DIAGRAM:

The integration system is divided into four major areas of research and development as seen in Figure 3. A complete discussion of each of these areas will be given later in this document.

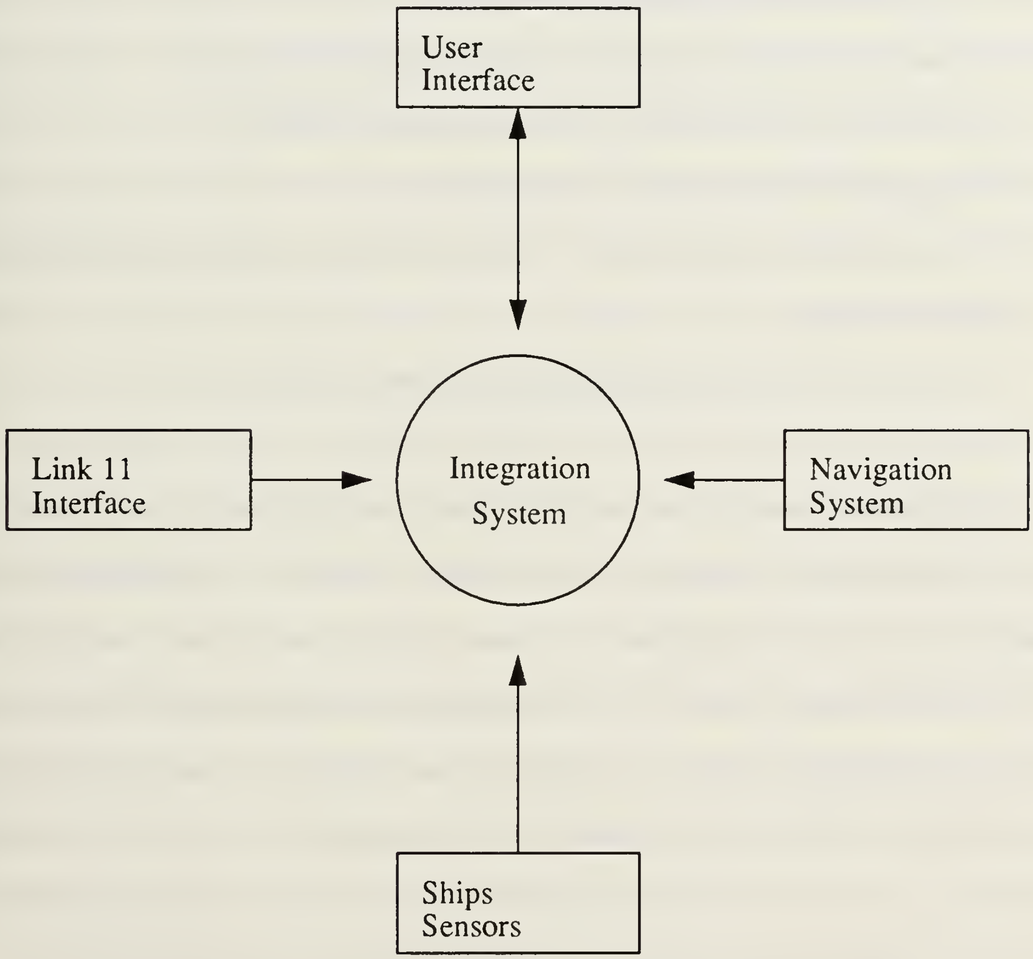


Figure 3 : LCCDS CONTEXT DIAGRAM

C. REQUIREMENTS FOR THE INTEGRATION SYSTEM

The requirements for the integration system appeared straightforward at first, but on closer examination we soon discovered that each of the more general requirements as outlined by Reference 1, Enclosure1 must be expanded to meet our specific needs. Listed below are the general requirements:

1. Use Ada as the implementation language.
2. Integrate an object-oriented Database Management System.
3. Integrate a manual tracking and identification capability.
4. Integrate a receive only link 11 capability.
5. Integrate an on ship navigation and maneuvering capability, along with display of shoreline maps.
6. Integrate an autotracking capability [Enclosure 1 of Ref. 1].

Expansion of these requirements is interlocked with the general design of the complete LCCDS. We began by looking at the qualities of Ada as the implementation language. Because of the Real_Time requirement for the LCCDS, parallel processing is a must.

The basic design feature of the Ada language is centered around the use of "Objects" for program design. An object is a data structure consisting of a unique identifier and an associated set of functions and procedures that operate on the object. This meaning of the term object may not be universally agreed upon, but it is our working definition, and will be used throughout the design of the integration system. These operators are the only allowed means of manipulating the object. A number of advantages follow from this design approach. Objects and their associated functions and procedures form a natural boundary along which to subdivide the integration system. Because the structure of a data type is

hidden from all but its associated operators, changes to the structure have a limited impact on the overall system. This feature greatly simplifies program modification and maintenance.

Ada provides a construct called a "Package" that allows the programmer to encapsulate objects and their associated functions and procedures. In addition, it allows for "private" types and "limited" private types that further restrict encapsulation so that objects of these types, while visible to the program parts, can only be manipulated by the functions and procedures it has referenced. A combination of these features permit the programmer to hide data structure implementation and create "abstract" data types. The use of the attribute *private* means that the programmer cannot use any knowledge of how the data type is to be implemented in the integration system. This allows for user changes in the basic features of the LCCDS but maintaining the integrity of the integration system. The integration system will take full advantage of each of these features.

Ada provides a "Task" construct, which is a feature that allows the programmer to divide a program into logically concurrent operations with synchronization between each or all of the operations. In addition to forming the basis for Real-Time operations, Tasks also provide a means of increasing processing efficiency in a parallel processor environment like the integration system for the LCCDS. Like packages, the task has a specification part and a body, however, the specification part is used solely to declare the synchronization point or entry point to the task. The entry point is used to indicate where the message is received or transmitted by the task.

The discussion of Ada packages and tasks would not be complete without an explanation of the Ada features "with" and "use". The with and use clauses are the mechanism by which the integration system environment is made available to all the elements contained within. The with clause tells the compiler that the programmer intends to use data types, procedures, and functions defined somewhere in the package specified.

The use clause tells the compiler that the programmer desires to reference the data types, procedures, and functions located somewhere in the package specified.

The use of data abstraction provides for the integration system several advantages:

1. A clearer conceptualizing of the problem or procedure being written and incorporated into the integration system.

2. More reliable data security.

3. A more reliable means of avoiding side effects.

4. Easier modification of the implementation as changes or updates occur.

Making use of or reuse of algorithms that have been implemented previously is a major advantage of program abstraction. Another advantage of this programming style is that it can be modeled more readily using mathematical techniques, thus opening up greater possibilities for correctness proof methods. Correctness proof is a major concern of the integration system since lives will depend on its effectiveness and precision.

The Ada language provides high level facilities for expressing concurrent algorithms parallel processes. These facilities are tasks, and along with subprograms, packages, and generic units, they constitute the physical unit make up of which our programs will be composed. Synchronization between any two of these task occurs when the task issuing an ENTRY call and the task ACCEPTING an entry call establish a rendezvous. The two tasks communicate with each other in both directions during this rendezvous.

Several task can rendezvous with each other, in groups of two or more, at any instant. If several tasks need to rendezvous with the same task, then these entry calls are placed in a queue associated with the entry and accepted in first in-first out order. By this method careful control of the tasks and their order of execution can be artificially established. By this method also we can set a system of priorities without using the Ada task specification "priority".

Deadlock is a concern: NO DEADLOCKS is a requirement for the integration system. Therefore it is absolutely essential to build deadlock prevention into the system. This idea is one that draws a large amount of concern and articles written on the subject. There are two basic fields of belief in the area, one is that deadlocks cannot be prevented and must be handled when they occur. The other is that deadlocks can be prevented and with careful planning and design, and that prevention is preferable to control. In our case, if a task(one) makes an entry call to a task(two) that is in the entry call queue of a task(three), which is in the entry queue of task(one), then deadlock occurs. The design of the system is such that this situation does not occur. Clearly, we have chosen to handle deadlocks by prevention, but have also considered controls and exceptions if the situation arises. Other methods and controls will be discussed later in the document. Research on formal methods and tools to ensure that designs are free from deadlocks is in progress [Ref. 24].

As a subunit within the integration system the database has only one type of object, Track. There does exist, however, several classes of the object. The database features space for unlimited instances of each class, limited only by the amount of swap space available to the workstation.

The integration system must provide a function by which the user can manually enter a track. Incorporated in this task will be provisions allowing the user to change certain attributes of the Track but, restricting these changes to Track identification number and other amplifying information.

The integration system must receive from the Global Positioning System ownship fix (Geographic_Position) data which consist of a Latitude, Longitude, and a Greenwich Mean Time(GMT). A Global_Position is the Latitude converted to an angle from the equator and the Longitude converted to an azimuth from the Greenwich Meridian. This data string must be translated and formatted into system data format. The ownship system data is to be stored in the database as track zero and used to define the ownship track. Ownship track is

used by the system to compute course, speed, closest point of approach, range, and bearing information on a user designated track.

The integration system must receive Link 11 data transmitted via the standard fleet UHF/HF communication channels. The data as received is a cryptogram and not usable by the integration system, therefore the data must be deciphered and translated into system format. To accomplish this translation we propose to use a system already being used in the fleet. This translation is a major project in itself and not a primary requirement for the prototype version of the LCCDS. The system proposed to translate the Link message input to M-series messages is the Link Monitoring System (LMS 11r) which receives the Link 11 data directly from the communications link and with a cryptographic unit (KG-40) in-line, translate the data into English M series messages which can be sent to the link 11 processor inside the integration system. The link 11 processor translates the M series messages to a string of system formatted characters representing a relative position from DLRP of each contact. The integration system will then store each of these contacts in the database as a track.

A subset of these tracks determined by a filtering process designated by the user, can then be graphically displayed. The filter system is a collection of individual filters that can be combined together by utilizing the mathematical expressions *and* and *or*. This combination filter acts as a single filter and forms a TACPLOT, which is used by the integration system to send to the user for graphic display those tracks and situations requested. Filters are discussed in more detail later in this document. The shoreline maps and auto tracking capability listed in the NAVSEA requirements are not a part of this research project.

D. INTEGRATION SYSTEM CONTEXT DIAGRAM

Figure 4 is the context diagram of the integration system. The diagram is used to illustrate the direction and paths of communication between the various elements of the integration system, the user interface, the link handler, and the navigation handler.

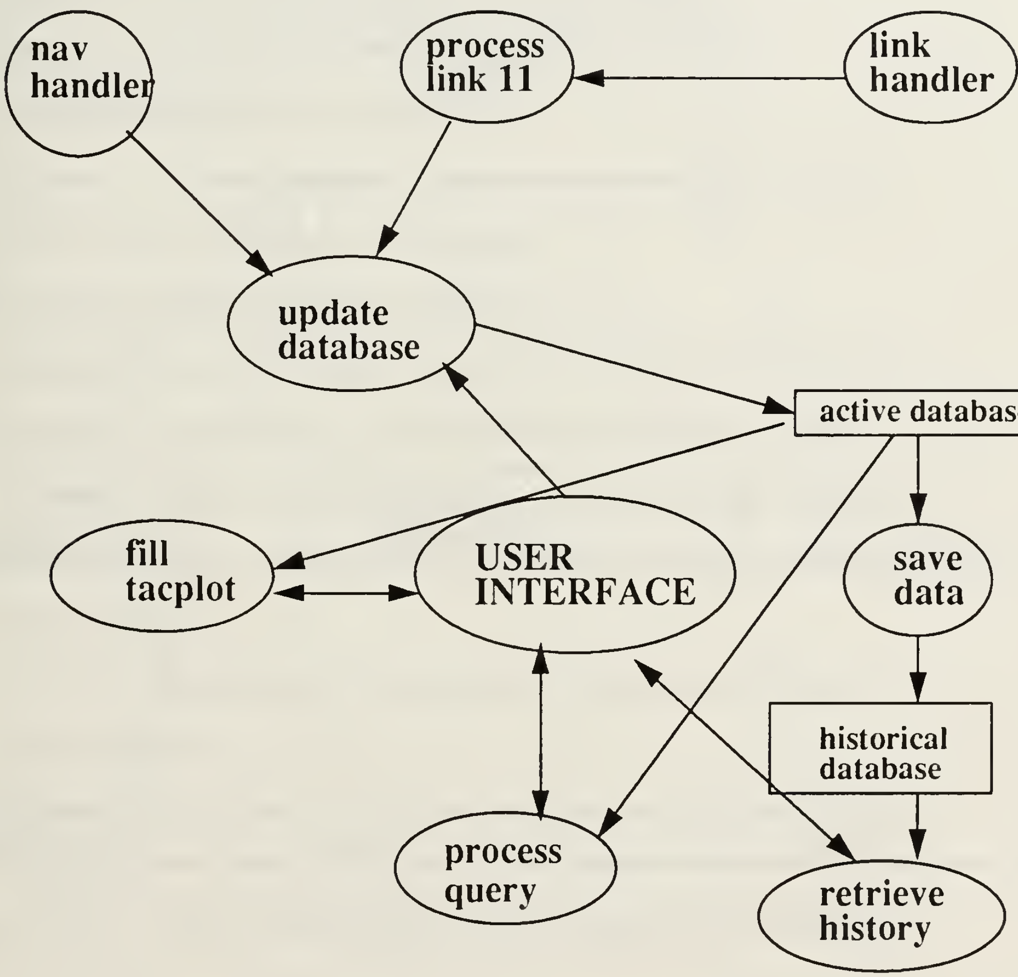


Figure 4 : INTEGRATION SYSTEM CONTEXT DIAGRAM

E. INTEGRATION SYSTEM STRUCTURE DIAGRAM

Figure 5 is the integration system structure diagram illustrating the individual sections or functions the integration system is naturally divided. Each section may contain several individual and unique functions or task which together accomplish the desired mission of that section.

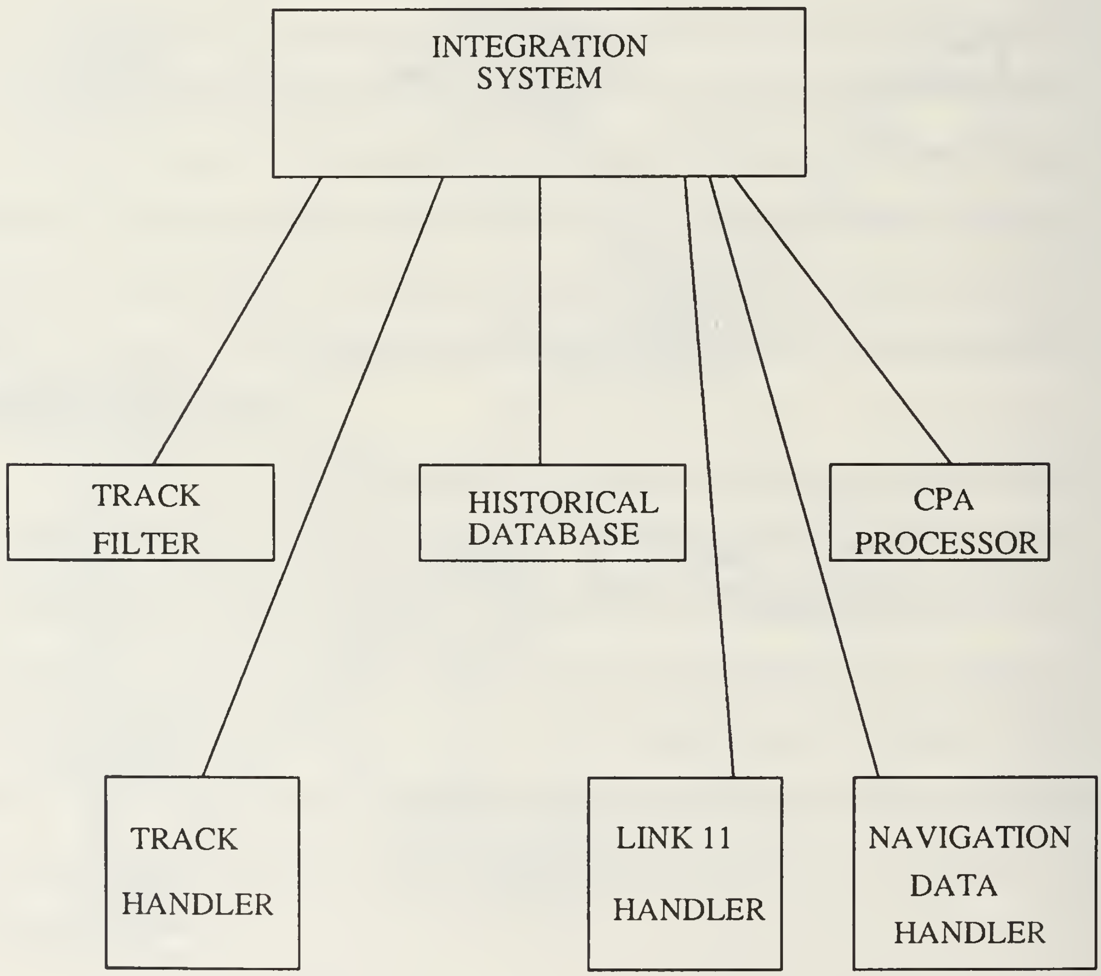


Figure 5 : INTEGRATION SYSTEM STRUCTURE DIAGRAM

F. EVENT LIST:

A list of external events that cause a response by the integration system is shown in Figure 6.

1.Simulus: Receive ownship data from the Navigation interface .

Response: Interpret and store ownship position(fix) in database.

2. Stimulus: Receive track data from Link 11 NTDS.

Response: Interpret and store NTDS tracks in database.

3. Stimulus: Receive filter from the user.

Response : Provide for graphic display of tracks specified by filter.

4. Receive new track data from the user.

Response: Interpret and store in database.

5. Stimulus: Receive a request to provide CPA data from the user.

Response: Interpret and provide forgraphic display CPA data.

6. Stimulus: Receive track information request from the user.

Response: Provide for the user track identification number and category of track specified.

7. Stimulus: Receive flag from navigation interface indicating loss of sensor signal.

Response: Provide user with alarm specifying loss of sensor signal.

Figure 6 : INTEGRATION SYSTEM EVENT LIST

List of events which will occur in the navigation system as a response to the action of one or more sensors are found in Figure 7.

1. Stimulus: Receive ownship fix data from the Global Positioning System.

Response: Translate GPS data to an Ada string of characters and transmit via communication link and RS 232 communication port to the integration system.

2. Stimulus: Receive ownship course from ships gyroscope.

Response: Transmit to integration system.

3. Stimulus: Receive water depth under the keel from ships fathometer.

Response: Transmit to integration system.

4. Stimulus: Receive ownship speed made good through the water from ships pitsword.

Response: Transmit to the integration system.

5. Stimulus: Receive contact information from the ships radar.

Response: Translate data to an Ada string of characters representing a global position and transmit to the integration system.

Figure 7 : NAVIGATION SYSTEM EVENT LIST

List of events that originate from the user or integration system and trigger a response from the user interface are found in Figure 8.

1. Stimulus: Receive updated tacplot from integration system.
Response: Provide graphic display of tracks specified by filter.
2. Stimulus: Receive update of track category and amplifying data from integration system.
Response: Provide graphic display of track category and amplifying data.
3. Stimulus: Receive track data from the integration system.
Response: Provide corrections to local tracks.
4. Stimulus: Receive CPA information from the integration system on any specified track and ownship track.
Response: Provide graphic display of CPA data.
5. Stimulus: Receive CPA information from the integration system on any two specified tracks other than ownship.
Response: Provide graphic display of CPA data.
6. Stimulus: Receive initialize the system from integration.
Response: User enters desired system setup.

Figure 8 : USER INTERFACE EVENT LIST

III. DESIGN OF THE INTEGRATION SYSTEM.

A. INTERFACE SPECIFICATION FOR THE INTEGRATION SYSTEM

One approach to the specification of concurrent programs is called behavioral. It starts by describing the possible events and actions, series of events and/or series of responses, in which part or all of a program may engage. The first big step was to take these descriptions and translate them into executable specifications. With this partial tool for designing concurrent programs, the construction of the integration system begin. At each level of the integration system we conducted a comparison of the different implementation methods available. Particularly noteworthy is that we found it readily easy to translate these implementation ideas into Ada code. More specifically, by using rendezvous and nondeterministic "Select" statements of Ada Tasking ensure the parallel processing we seek.

The integration system shall provide detailed information on all aspects of the tactical situation and system control, operating parameters and status. This information is obtained from the Tactical Database which shall be an object-oriented database management system written in Ada. The system will provide a flexible, easy to use, window based user interface. A navigation interface will provide the system with ownship information and track data, as well as navigation data.

B. STATEMENT OF PURPOSE

The purpose of the integration system of the LCCDSWS is to integrate the user interface, the navigation system, the receive-only link interface 11 and the object- oriented database management system. The system is to maintain and display a real time picture of the tactical environment for the specific platform on which the system is located.

The results of this integration will store in the database all tracks, including the ownship track which includes ownship Navigation and Maneuvering data. The integration system will use filters provided by the user to determine the contents of the tacplot which is sent to the user interface for display.

C. CONSTRAINTS

Software development for Department of Defense must adhere to Department of Defense Military Standard 2167-A Defense System Software Development, 29 February 1988[Ref. 25], Department of Defense Military Standard 2168 Defense System Software Quality Program, 29 February 1988[Ref. 26], and American National Standard Institute Military Standard 1815A-1983 Reference Manual for the Ada Programming Language, 17 February 1983[Ref. 27].

Specified in the Requirement Analysis [Ref. 3] Seveney and Steinberg thesis, are the LCCDSWS, prototype constraints. These constraints will be used as a guideline for the constraints definitions of the integration system. The performance constraints may be evaluated at several different levels and in several different contexts but we will focus on a limited view from the standpoint of the integration system only.

1. Resource constraints: The basic resources are available in the LCCDS team and in the faculty and staff of the Naval Postgraduate School.

2. Implementation constraints: Hardware available is the Suns Microsystems Sparcstation 2 machine. The system is configured in a stand-alone unit configuration with four each RS-232 communication ports used for interface with the Link, GPS, and ships sensors. Operating System as defined in reference 2 is derived from the UC Berkely Version 4.2 BSD and Bell Lab's UNIX system version 32v [Ref. 32].

In accordance with Reference 1 and The Department of Defense policy the implementation language for the system will be Ada. In this particular application Verdix Ada 6.0 is used.

3. Performance constraints: Performance for the LCCDS workstation include Real_Time data processing and display. In this application system performance has an upper bound: Reference 1, Enclosure 1 defines Real_Time to mean that response time must be less than or equal to four seconds.

D. THE INTEGRATION SYSTEM

The design of the LCCDS is not that of an embedded system, however, the integration system contains functions and procedures not visible to the user. These functions and procedures, in some cases found in Ada Tasks, perform a vital role in the overall systems response and behavior. The design of the integration system as a Real_Time embedded system requires the use of parallel processing.

In order to meet the time constraints specified in Reference 1, special attention must be given to the order and magnitude of the Ada programs and packages which make up the integration system. The integration system is the main processing element of the LCCDS. Other elements such as the navigation system, Link 11, ships sensors have a one way communication link and only provide data to the integration system. The user interface element has a two way communication link with the Integration System, but is used to display, retrieve, and add to the data already in the system. The integration system stores the data received from these sources in the active database (located in RAM). The data is stored in a data structure called Track, which is defined in the database section of this document.

The system as configured can retrieve the data to perform various operations and functions on Track as required by the user or predefined by the system. The requirements for the system, list a number of these operations and functions [Ref. 1, 2].

1. Provide a filtered set of tracks to the user interface for graphic display.

2. Provide the user with the ability to select the category and type of track to be displayed.

3. Provide the user with closest point of approach data between any pair of tracks selected by the user.

4. Track position to be dead reckoned using current track bearing and speed.

5. Allow the user to make changes to tracks in the database.

6. Provide the user with safe maneuvering data.

The integration system receives track data from three sources:

1. Manual input from the user as illustrated in Figure 9.

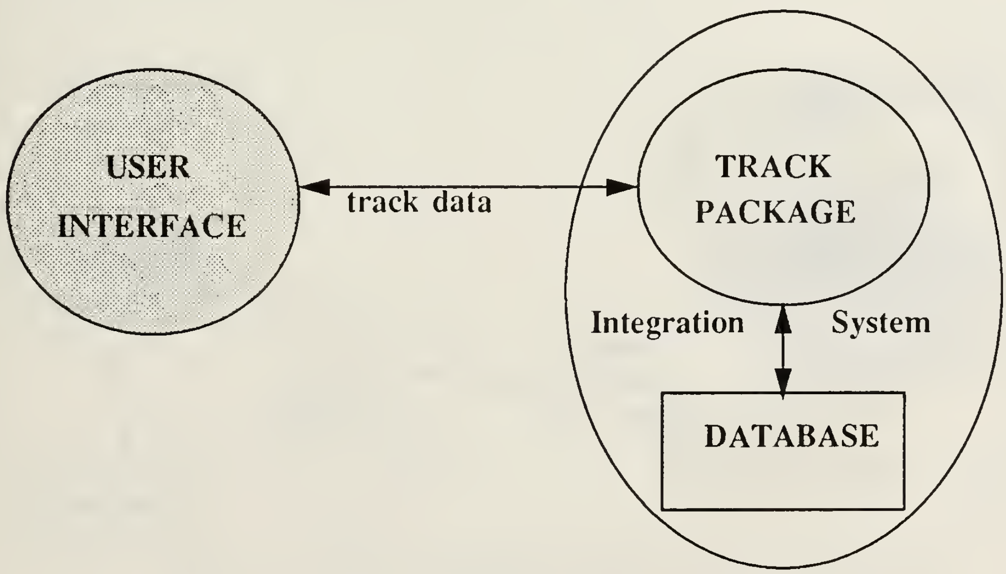


Figure 9 : TRACK INPUT BY USER

2. Via communications interface with link 11 as illustrated in Figure 10.

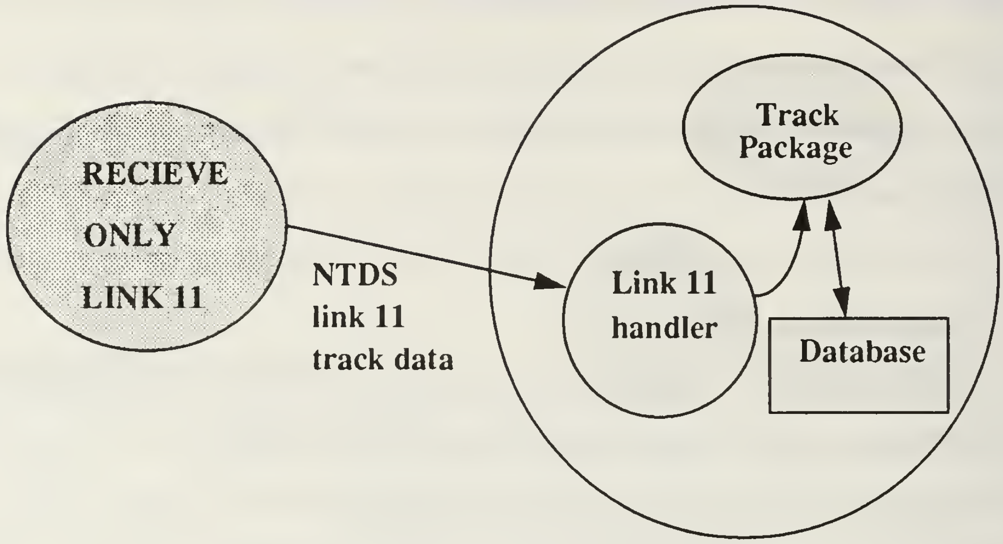


Figure 10 : TRACK INPUT BY LINK 11

3. Via communications interface with the ships sensors(radar) as illustrated in Figure 11.

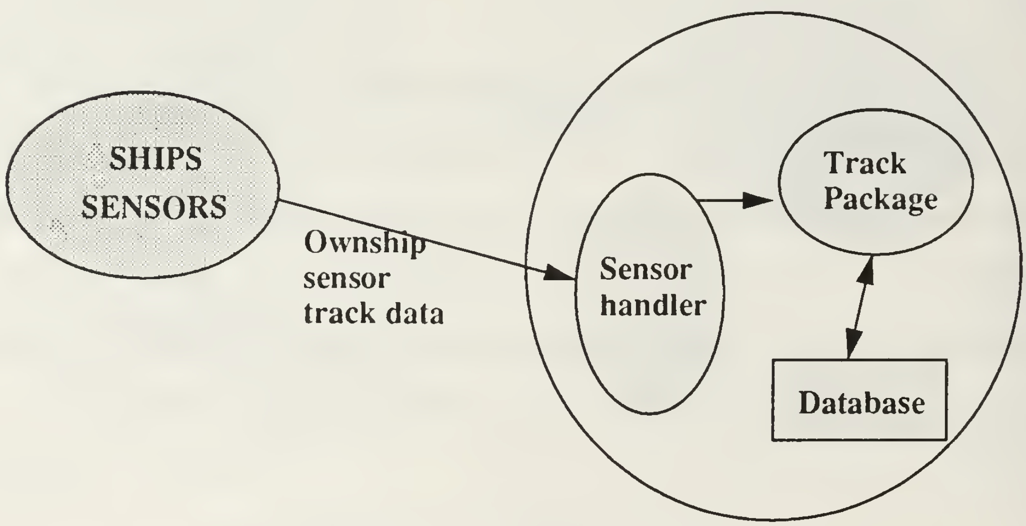


Figure 11 : TRACK INPUT BY OWNSHIP SENSOR

The prime objective of the LCCDS is to provide a clear and concise tactical picture for the ship commander. This tactical picture must be presented in a manner which accurately represents the tactical problem (situation) comprehensibly to the user. The integration system allows the user freedom to concentrate on the situation via user predefined filters. Regardless of the mission or tactical situation, a ships sensors provide only raw data. Even when this data is graphically displayed relative to ownship, it is still only useful when the user applies intelligence to the overall situation.

A simplified view on the process of collecting, filtering, and displaying tactically significant data in a Real Time environment is in Figure 12.

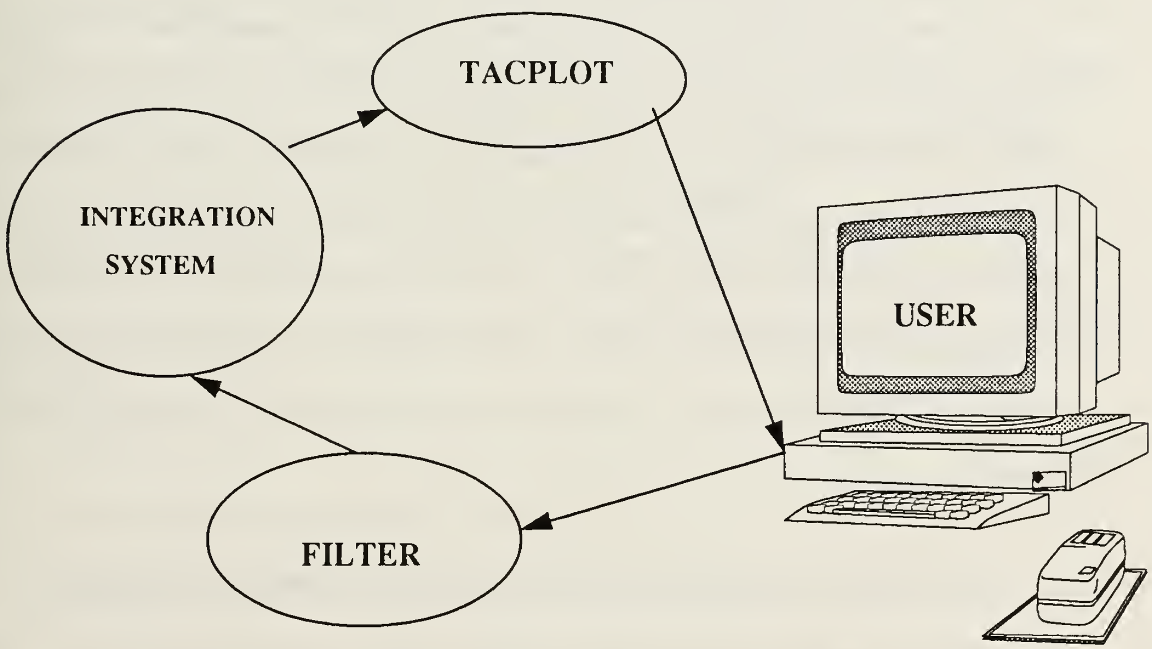


Figure 12 : TRACK FILTER STRUCTURE DIAGRAM

The integration system provides navigation, link 11, and user interface inputs to the database. The input is not direct, but through the integration system, allowing for control of the data stored in the database. A specific package is written inside the integration system to interface with the navigation system. The navigation system package provides a facility for converting GPS data into an ownship track. The link 11 package processes the link 11 tracks and after filtering the track base, stores all accepted tracks in the database. Local or user generated tracks is part of the track package.

The integration system consist of a main Ada task that makes entry calls to the various tasks, functions, and procedures that collectively makeup the integration system. The simplified function or purpose of the integration system is to receive data from various sources and translate/parse this raw data input into data the user_interface can use for graphic display and store a duplicate set of data in the database.

The user has available a set of options by which to manipulate the system filter algorithm. The user may select a single atomic filter or a series of atomic filters and by applying the mathematical *and* and *or* statements combine these filters to create a single *and* filter. This single *and* filter provides a template which the integration system uses to retrieve only tracks that meet the specific properties of the Tacplot. The Tacplot filed with the tracks that meet the filter are sent to the user interface for graphic display of the tactical situation as illustrated in Figure 12. How the data is displayed is not a consideration of the integration system.

E. THE OBJECT ORIENTED DATABASE MANAGEMENT SYSTEM

The requirements for the LCCDS specify design and implementation of an object-oriented database system. The purpose of this database is to manage the tactical information store of the LCCDS. The information is used to display a tactical picture of a ship's local environment and provide pertinent answers to queries defined by the user. The data

structure and methods of the database, as well as the supporting software components are to be implemented in Ada. The features included in our database are based on the following considerations:

1. Real_Time performance: Safety and Maneuverability of the ship, as well as tactical decision-making demands Real_Time performance.

2. Maintainability: Using an object-oriented approach to the database ensures the methods and procedures defined on an object will not be affected if the data structure representing the object requires alteration.

3. Transaction concurrency: In order to maintain Real_Time performance, parallel execution of separate tasks must occur. The parallel processing of these tasks, however, introduces potential of deadlock situations that should be prevented.

The design of our database responds to the above considerations utilizing:

1. Variant Ada records [Ref. 33] to define a single common object class. The main data structure holding the instances of the defined objects allows for rapid retrieval and ease of updating. Locking protocols prohibits conflicting transactions on the database.

2. Ada tasks to handle the transaction concurrency problem.

We start our explanation of the record structure by defining the catalog, also known as the database description or schema [Ref. 5, 10]. The catalog contains the following information:

1. The constraints.

2. Usage standards and application programs.

3. Descriptions and user information.

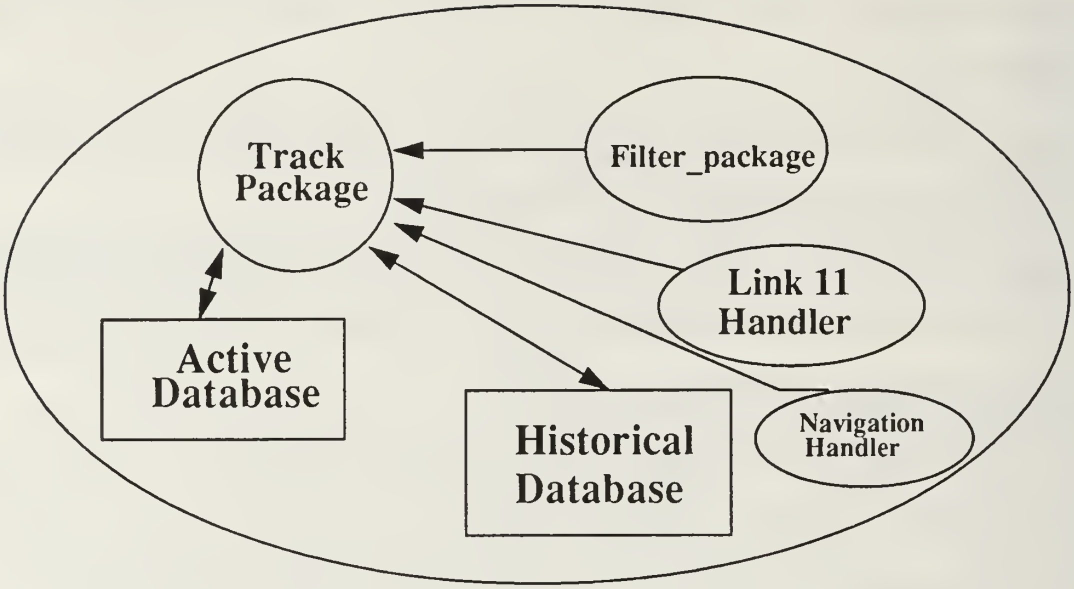


Figure 13 : DATABASE COMMUNICATIONS DIAGRAM

The system provides the user with the ability to find a specified track in the database, add a track, alter a track, drop a track, send a track to history, restore an altered track to database, see Figure 13.

As discussed previously the Global Positioning System (Trimble-4000 S) illustrated in Figure 14 transmits the current fix data of ownship to the navigation handler via an RS-232 communication port via an RS-232 communication port. The navigation handler parses this data and translates it to a LCCDS usable format. The integration system receives from the navigation handler a string of characters which represent the position of ownship at a specific time.

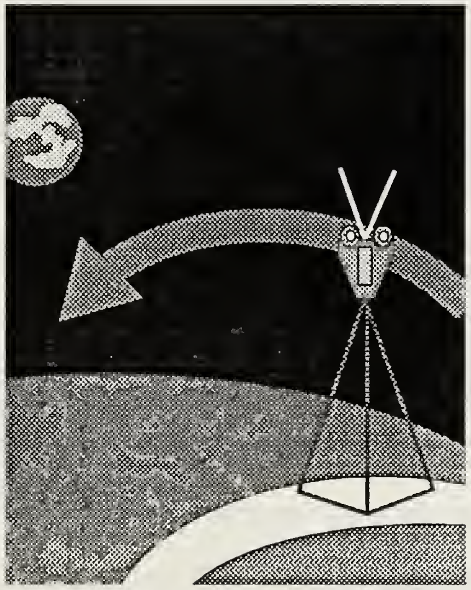


Figure 14 : GLOBAL POSITIONING SYSTEM

The string of characters is parsed and converted into a Global_Observation for ownship. Data is received from the GPS at one second intervals. The navigation handler stores each of these data_input_strings in a buffer ready for the integration system to read. When the integration system makes a request to read data the navigation handler locks the buffer and does not allow the GPS to perform its normal one second overwriting of the data in the buffer with new data.

When the integration system has completed the read function the navigation handler unlocks the buffer and allows the GPS to overwrite the buffer with the next full string of data. An interval of every four seconds is required for the integration system to update the ownship Global_Observation.

Link 11 tracks are received by the system and converted to the track type. The system stores the tracks in the database. Filtering of these Link 11 tracks occurs in two stages, first as the tracks are received and deciphered, the second when the user designed filter is used to fill the Tacplot for graphic display of tracks.

F. THE LINK 11 RECIEVE ONLY SYSTEM

A vital feature in the LCCDS is the ability to receive all contact information reported by the task force on the NTDS Link 11. The data gathered and displayed from this source will give the Commanding Officer a clear tactical picture of all elements in the force. The Link provides a measure of security for ships maneuverability and tactical defense. This study did not consider a two-way communication link because the value of two-way communication to a non-combatant ship is unclear. However, data from ownship sensors could be useful to other combatant ships.

we propose to utilize software and hardware from an outside source to translate the NTDS Link 11 data into source code the system can use. The Link 11 interface with the integration system consists of the link 11 handler designed inside the integration system and communicating directly with it is the external Link 11 data translator and decoder. The link handler is an Ada function which breaks a string of characters into the individual parts of the data type Track and stores the array of parts in a buffer waiting for the integration system to lock the buffer and read out the data. After reading the contents of the buffer the integration system unlocks the buffer. The link handler then repeats the process.

Once this translator package is in hand, we can proceed to design an Ada package capable of parsing the NTDS Link 11 code string, M messages, and breaking them into there individual elements. Once the individual elements are available the system can convert them into a Track record. A new LCCDS track number is be assigned to each track with a pointer from the NTDS track number to its associated system track number. The Track record is be stored in the active database as a track. We limited our work on the link handler to developing a specification.

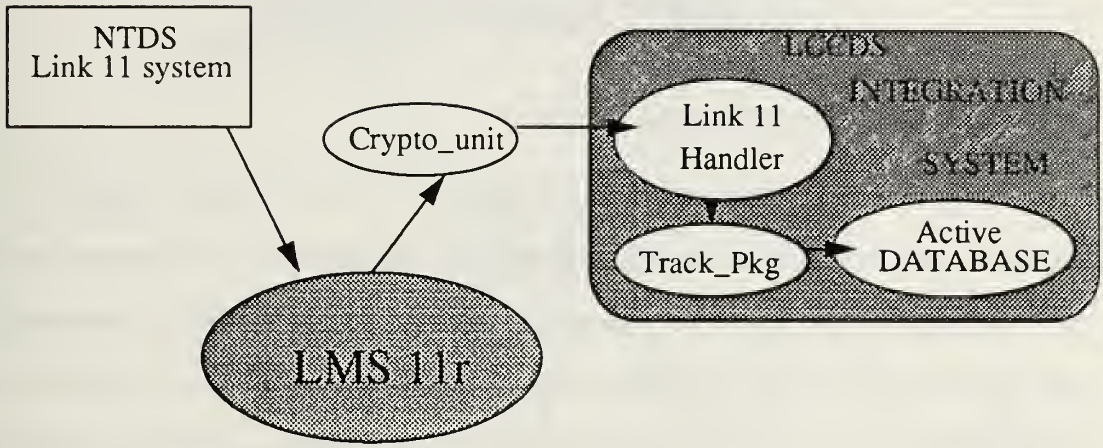


Figure 15 : LINK 11 RECIEVE ONLY CONTEXT DIAGRAM

The system recommended to decipher and translate the Link 11 data into a string of correct LCCDS message format characters is the Link Monitoring Set 11 r (LMS 11r) system as illustrated in Figure 15. The LMS 11r system is a Link 11 receive only Data Terminal Set which can provide a continuous sting of two each sixteen bit parallel messages of the Link 11 data. These messages are then passed through the crypto- unit (KG-40 for LOS - UHF/HF and KG-84 for SATCOM - UHF)) which decodes the messages to M series messages. Using the format prescribed in OP-SPEC 411.2 these M series messages can be translated in the system format (English Text) by the integration system link 11 processor package. Because of the classification (CONFIDENTIAL) of the link 11 material a removable hard drive or tape drive is recommended for secondary memory. At this point a discussion of the protocol for Link 11 data receipt, MIL-STD-1397 input data, would be appropriate if this research paper was classified. Because the paper is unclassified we will leave this discussion to the follow-on research and development of the Link 11 receive only system.

IV. INTEGRATION SYSTEM /OODBMS ARCHITECTUAL DESIGN AND IMPLEMENTATION

A. INTEGRATION SYSTEM MODEL

The integration system software is designed as a set of Ada packages. This concept allows for greater versatility and application of the Ada programs and functions developed. The integration system provides navigation, link 11, and user inputs to the database. The input is not direct, but through the integration system, allowing for control of the data stored in the database. A specific package is contained in the integration system to interface with the navigation system.

A general discussion of the packages and how we applied them to the overall design concept of the integration system follows Figure 16 which is a package dependency diagram of the integration system. In Figures 16 and 17 nodes are Ada packages, and the arrows depict Ada *with* statements.

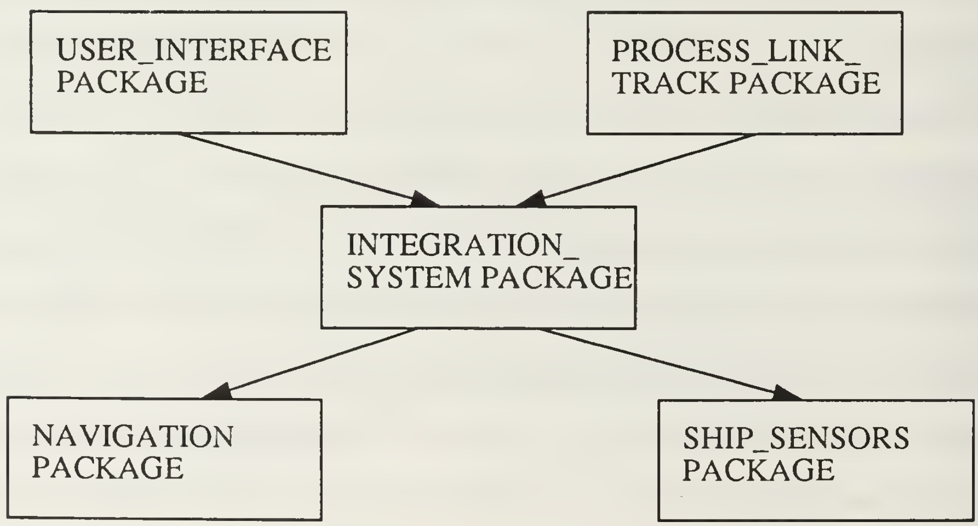


Figure 16: PACKAGE DEPENDENCY DIAGRAM LEVEL 0

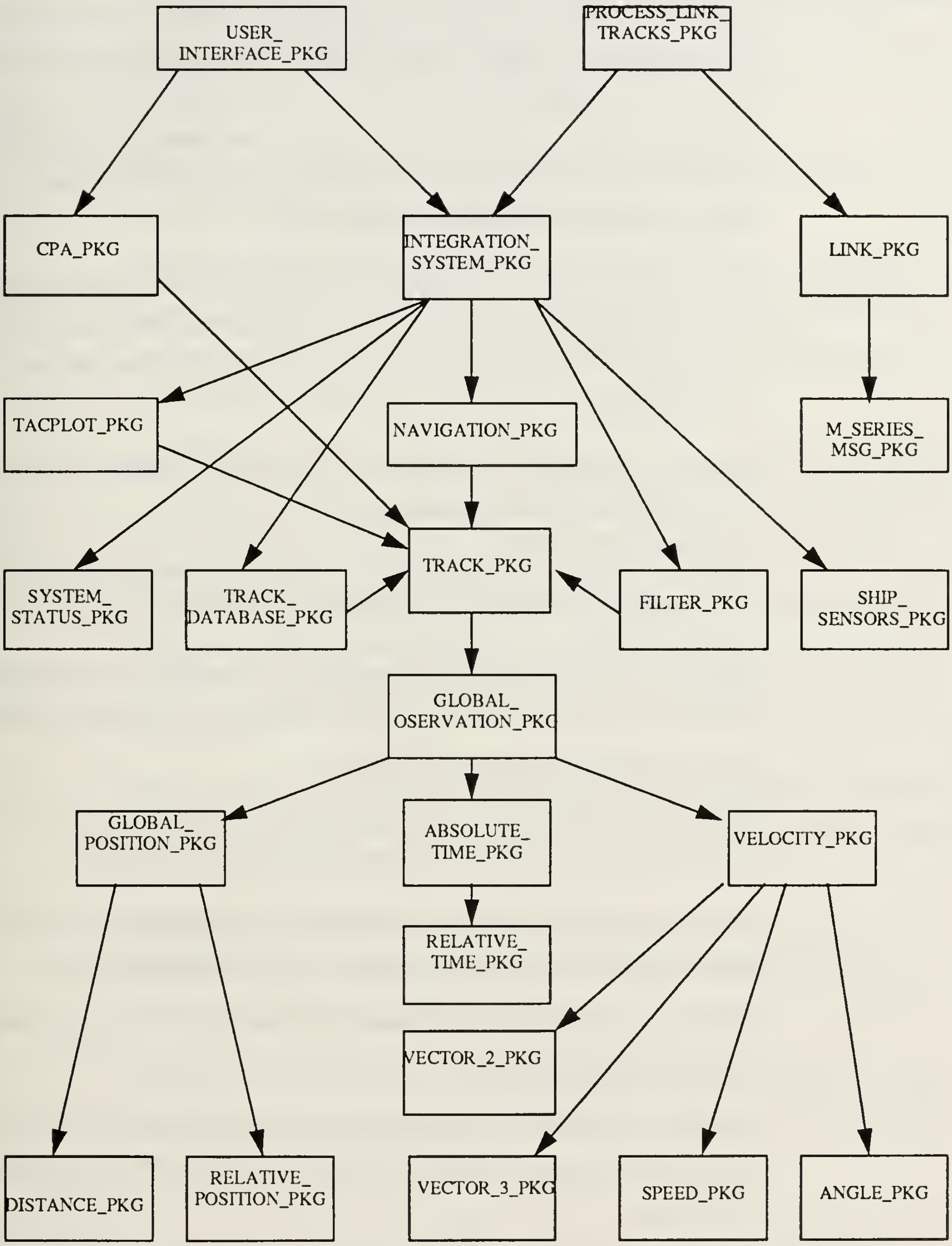


Figure 17: PACKAGE DEPENDENCY DIAGRAM

1. LIST OF INTEGRATION SYSTEM PACKAGES:

- **INTEGRATION SYSTEM PACKAGE**: The purpose of the package is to receive data or information from various sources, translate/parse the raw data input into integration system formatted data, store the data in the database as a track, and send the data to the user_interface for graphic display of the tactical situation. The package also performs time synchronization functions for external tasks.
- **FILTER PACKAGE**: The purpose of the package is to represent policies for choosing which tracks are entered in the database and which are shown on the graphic display. The policies are defined by the user via the user_interface.
- **TRACK PACKAGE**: The purpose of the package is creation, deletion, and modification of tracks in the database.
- **CPA PACKAGE**: The purpose of the package is computation of the closest point of approach between any two tracks specified by the user.
- **VELOCITY PACKAGE**: The purpose of the package is to represent the velocity of a specified track. Velocity is defined as a two dimensional vector, representing course and speed.
- **VECTOR_2 PACKAGE**: The purpose of the package is to provide a means of using two dimension vectors for various applications.
- **VECTOR_3 PACKAGE**: The purpose of the package is to provide a means of using three dimension vectors in various applications.
- **SPEED PACKAGE**: The purpose of the package is to represent speed in knots or yards per second.
- **ANGLE PACKAGE**: The purpose of the package is to offer a means of representing an angle in radians or degrees and functions to return attributes of the angle.
- **DISTANCE PACKAGE**: The purpose of the package is to offer a means of representing distance in yards or nautical miles.

- **ABSOLUTE TIME PACKAGE**: The purpose of the package is to provide the integration system constant access to system time. Defines the abstract data type Absolute_Time and associated functions. System time can be displayed as Greenwich Mean Time or Local Mean Time depending on user needs.

- **RELATIVE TIME PACKAGE**: The purpose of the package is to represent the length of the (interval)between two events.

- **GLOBAL POSITION PACKAGE**: The purpose of the package is to represent geographical positions on the earth. Input and output in terms of latitude and longitude are provided. Internally uses an angle from the equator and an angle from the Greenwich Meridian.

- **GLOBAL OBSERVATION PACKAGE**: The purpose of the package is to represent a global_observation(global_position, velocity, and time) for a track. The global observation indicates current position of the track.

- **RELATIVE POSITION PACKAGE**: The purpose of the package is to compute the bearing and range of a track from a reference track. Bearing is defined as an angle from true north and range is the distance between the two tracks.

- **RELATIVE OBSERVATION PACKAGE**: The purpose of the package is to define a data type Relative_Observation that stores a Relative_Position and an Observation_Time.

- **TRACK DATABASE PACKAGE**: The purpose of the package is to provide a means to store the tracks in the system. To accomplish this the package creates a linked list of tracks.

- **LINK PACKAGE**: The package converts M series messages into system formatted tracks. These tracks are stored in the database as link controlled tracks.

- **NAVIGATION PACKAGE**: The purpose of the package is to keep track of ownship position via a communication port that accepts global positioning system data. The received data is translated into integration system track format and stored in the database as ownship current location.
- **SYSTEM STATUS PACKAGE:** The purpose of the package is to provide the system with a means to enable or disable the communication link between the system and the ships sensors. The package provides the integration system with a means of indicating a up and operating or down and off status of the ships sensors.
- **M SERIES MSG PACKAGE**: The purpose of the package is to provide a means of activating a communication port to read in the link M series messages from the LMS 11r and storing the messages in a buffer.
- **PROCESS LINK TRACKS PACKAGE**: The purpose of the package is to read from the buffer each M series message. Using the LINK package procedures/functions, each M series message is converted to an integration system link track. The LINK tracks are processed as integration system tracks and stored in the database.

2. ABSTRACT DATA TYPES:

a. TRACK

(1) Description: A TRACK represents the observations and descriptions of a tactically significant contact. The implementation of the TRACK type is given in Appendix C, p. 103. There are several different kinds of TRACKs; each of which is identified by its TRACK_CATEGORY (see Function TRK_CATEGORY). The possible values of TRACK_CATEGORY are:

a. SURFACE_PLATFORM: In nautical terms, a surface platform is defined as any man-made vessel designed to operate on the surface of the water. For a more detailed definition refer to Reference 2.

b. SUBSURFACE_PLATFORM: In nautical terms, a subsurface platform is defined as any man-made vessel designed to operate below the surface of the water. For a more detailed definition refer to Reference 2.

c. AIR_PLATFORM: An air platform is any man-made object designed to operate above the earth's surface. The platform has an ALTITUDE. For a more detailed definition refer to Reference 2.

d. UNKNOWN: An unknown TRACK_CATEGORY is defined as any TRACK whose TRACK_CATEGORY has not yet been established by the user.

The TRACK_CATEGORY of an UNKNOWN TRACK can be changed via the operation CHANGE_TRACK_CATEGORY.

e. REGION: REGIONs consist of two types, CIRCLE and POLYGON. A REGION is stored in the database as a TRACK. A CIRCLE contains a center (GLOBAL_POSITION) and a radius (DISTANCE). A POLYGON contains from three to twenty vertices (GLOBAL_POSITIONS) that form the POLYGON. The REGION may be relative to a GLOBAL_POSITION which does not have motion or relative to a TRACK that has VELOCITY. A REGION may represent an operating area in which the platform operates or may represent a restricted area in which platform movement is constrained or forbidden.

f. PATH: A PATH consists of a series of WAYPOINTs (GLOBAL_POSITIONS) and is stored in the database as a TRACK. A time is assigned to each WAYPOINT and represents a desired time to arrive at the WAYPOINT. The array is passed to the user_interface for graphic display upon request. PATHs can be used to represent Path of Intended Movement(PIM) along which the platform travels. A PATH can be stored in history for later reference.

g. MAN_IN_WATER: A GLOBAL_POSITION used to mark the geographic location of a man lost overboard.

h. SPECIAL_POINT. A SPECIAL_POINT TRACK is defined as a single object, real or imaginary, man-made or natural, and not otherwise designated as

surface platform, subsurface platform, air platform, or unknown. A SPECIAL_POINT TRACK is further defined by its SPECIAL_POINT_CATEGORY. The possible values of a SPECIAL_POINT_CATEGORY are NAV_HAZARD, WAYPOINT, or GENERAL. All SPECIAL_POINT TRACKs have, as attributes, VELOCITY, and (GLOBAL_POSITION). A WAYPOINT is generally defined as an imaginary point at a specific GLOBAL_POSITION with an additional attribute TIME_TO that defines OWNSHIP's expected/desired arrival time to the WAYPOINT. A NAV_HAZARD is a SPECIAL_POINT that represents a physical object whose size and/or location presents a real hazard to navigation. A GENERAL SPECIAL_POINT is a SPECIAL_POINT not otherwise designated as a WAYPOINT or NAV_HAZARD. Its description may be elaborated in the TRACK's AMPL_INFO.

(2) Attributes: The following are attributes of TRACK:

a. Function TRACK_ID_NUMBER (TRK: TRACK) return NATURAL;

TRACKs are uniquely identified by their TRACK_ID_NUMBER. TRACK_ID's are unique throughout a mission, to make sure that the historical record is unambiguous. Every TRACK has a TRACK_ID_NUMBER regardless of its TRACK_CATEGORY. The TRACK_ID_NUMBERs are generated by the TRACK_TYPE and are a one up count process (see the variable TRACK_ID in the private part of the package TRACK_PKG specification. The correspondence between Link TRACK_ID's and TRACK_ID_NUMBER is maintained by the LINK_TABLE data structure in the package LINK_PKG.

b. TRACK location:

Function CURRENT_POSITION (TRK: TRACK) return GLOBAL_POSITION;

CURRENT_POSITION returns the GLOBAL_POSITION of the TRACK's dead-reckoned position from the last GLOBAL_OBSERVATION

Function RELATIVE_BEARING (REFERENCE_TRACK, TARGET_TRACK: TRACK) return ANGLE;

Returns the bearing angle from the course of the REFERENCE_TRACK to the TARGET_TRACK.

Function TRUE_BEARING (REFERENCE_TRACK, TARGET_TRACK: TRACK) return ANGLE;

Returns the bearing angle from true north to the TARGET_TRACK.

Function MOST_RECENT_OBSERVATION (TRK: TRACK) return ANGLE;

Returns the TRACK's last entered GLOBAL_OBSERVATION.

c. TRACK motion:

Function TRUE_VELOCITY (TRK: TRACK) return VELOCITY;

Returns TRACK's true course and speed relative to the surface of the earth as calculated in its MOST_RECENT_OBSERVATION.

Function TRUE_COURSE (TRK: TRACK) return ANGLE;

Returns TRACK's true course calculated in its MOST_RECENT_OBSERVATION.

Function TRUE_SPEED (TRK: TRACK) return SPEED;

Returns TRACK's true speed calculated in its MOST_RECENT_OBSERVATION.

Function TRACK_RELATIVE_VELOCITY (REFERENCE_TRACK, TARGET_TRACK: TRACK) return VELOCITY;

Returns TARGET_TRACK's relative motion (course and speed) relative to the given REFERENCE_TRACK.

Function RELATIVE_COURSE(REFERENCE_TRACK, TARGET_TRACK: TRACK) return ANGLE;

Returns TARGET_TRACK's relative course as seen from the reference TRACK.

d. TRACK intelligence information:

Function AMPL_INFO (TRK: TRACK) return AMP_STR.VSTRING;

Returns a string of characters that more clearly defines the identification or mission of the platform represented by the TRACK.

Function TRACK_IDENTITY (TRK: TRACK) return IDENTITY_TYPE;

Returns the TRACK's IDENTITY_TYPE, which can have the values UNKNOWN, FRIENDLY, HOSTILE, NEUTRAL.

Function PLATFORM_CLASS (TRK: TRACK) return V_AND_C_STR.VSTRING;

Returns a string of characters that define the class of the contact. Examples are Cruiser or Aircraft carrier.

Function VESSEL_NAME (TRK: TRACK) return V_AND_C_STR.VSTRING;

Returns a string of characters that represent the platforms name. An example is USS EDSON.

(3) Creation Operations A TRACK object is created by procedure CREATE_TRACK Appendix C, p. 130. A required parameter for this operation is, understandably, its first GLOBAL_OBSERVATION.

(4) Update Operations The package, TRACK_PKG, contains numerous functions and procedures to modify/update the attributes of TRACK objects as described in Reference 2.

b. FILTER

(1) Description: A FILTER is a predicate on TRACKs that defines a subset of all possible TRACKs. FILTERs are used to represent display policies. They describe a set of characteristics that a TRACK must possess in order to be graphically displayed.

Complex FILTERs are defined in terms of simpler AND_FILTERs. A FILTER predicate is a disjunction (or) of one or more AND_FILTERs; that is, if a TRACK meets all requirements of at least one of the AND_FILTERs, it is accepted for display. AND_FILTERs are composed of simpler ATOMIC_FILTERs. An AND_FILTER predicate is a conjunction (and) of zero or more ATOMIC_FILTERs; a TRACK satisfies an AND_FILTER if it meets all requirements of its component ATOMIC_FILTERs. Each ATOMIC_FILTER defines a single relational constraint on a TRACK. The implementation of the FILTER type is given in Appendix D, p. 153.

(2) Attributes: ATOMIC_FILTERs have the form [FILTER_CATEGORY RELATION CONSTANT]. The possible values of FILTER_CATEGORY are DISTANCE_FILTER, TRACK_CATEGORY_FILTER, and PLATFORM_IDENTITY_FILTER.

a. DISTANCE_FILTER describes a TRACK's distance from a reference TRACK or the TRACK's altitude (if air).

b. TRACK_CATEGORY_FILTER describes a TRACK's TRACK_CATEGORY.

c. PLATFORM_IDENTITY_FILTER describes a TRACK's IDENTITY_TYPE (UNKNOWN, HOSTILE, FRIENDLY, NEUTRAL).

d. RELATION identifies the FILTER_CATEGORY's relation to the input CONSTANT. The possible values of a RELATION are EQUAL, NOT_EQUAL, LESS, LESS_OR_EQUAL, GREATER, and GREATER_OR_EQUAL. An example ATOMIC_FILTER is "TRACK_CATEGORY EQUAL SURFACE_PLATFORM." This means that one requirement (ATOMIC_FILTER) of an AND_FILTER is that the TRACK must be of TRACK_CATEGORY SURFACE_PLATFORM.

(3) Creation Operations: ATOMIC_FILTERs are created through calls to either: MAKE_DISTANCE_ATOMIC_FILTER, MAKE_TRACK_CATEGORY_ATOMIC_FILTER, or MAKE_PLATFORM_IDENTITY_ATOMIC_FILTER.

Following the creation of an ATOMIC_FILTER, it is appended to its parent AND_FILTER through a call to ADD_ATOMIC_FILTER_TO_AND_FILTER. Once an AND_FILTER has been fully defined, it is appended to the FILTER through a call to ADD_AND_FILTER_TO_FILTER.

(4) Update Operations: FILTERs are updated as a result of the addition of AND_FILTERs. Once the FILTER is filled, the contents of that FILTER are unchangeable, unless a new FILTER is created, thus deleting the old ATOMIC_FILTERs and AND_FILTERs.

c. *TRACK_DATABASE*

(1) Description: TRACK_DATABASE represents the LCCDS database of TRACKs. The implementation of the TRACK_DATABASE type is given in Appendix Q, p. 231.

(2) Attributes: ACTIVE_TRACK(TRACK_DATABASE) returns a boolean value that tells whether or not a TRACK is active in the database. For example, following a call to FIND_TRACK_IN_DBASE(TRACK_ID), the function ACTIVE_TRACK(TRACK_DATABASE) will return FALSE if the TRACK was not found. Active relates to a TRACK receiving periodic updates by the function ADD_TRACK_OBSERVATION.

(3) Creation Operations: LCCDS contains one, and only one, object of type TRACK_DATBASE that is created at system initialization.

(4) Update Operations: TRACK_DATABASE is updated when a TRACK is added to the database (ADD_TRACK_TO_DBASE), when a TRACK is deleted from the database (DROP_TRACK_FROM_DBASE), and when the entire database is deleted (PURGE_ENTIRE_DBASE).

d. *GLOBAL_POSITION*

(1) Description: A GLOBAL_POSITION represents the earth coordinates of a TRACK geographic location. The implementation of the GLOBAL_POSITION type is given in Appendix N, p. 219. Internally we use a right-handed coordinate system centered on the center of the earth. The z axis points to the north pole, and the x axis points to the intersection of the equator and the Greenwich Meridian.

(2) Attributes: The geographic location is defined as a latitude and longitude of the TRACK. Latitude is defined as an angle from the equator (PHI) and Longitude is an angle from the Greenwich Meridian (THETA). GET_LATITUDE(GLOBAL_POSITION) and GET_LONGITUDE(GLOBAL_POSITION) are attributes of GLOBAL_POSITION that refer to latitude and longitude, respectively. A GLOBAL_POSITION, as used in LCCDS, cannot be changed once created. Its value can, however, be retrieved for use in the computations of other values.

(3) Creation Operations: The operations that create a GLOBAL_POSITION are MAKE_GLOBAL_POSITION and FIND_GLOBAL_POSITION. MAKE_GLOBAL_POSITION accepts the numerical equivalents of degrees, minutes, and seconds, as well as the latitude and longitude hemisphere identifiers and returns a GLOBAL_POSITION in terms of PHI and THETA. FIND_GLOBAL_POSITION returns a calculated GLOBAL_POSITION based on a RELATIVE_POSITION from another GLOBAL_POSITION.

(4) Update Operations: None

e. *LINK_TYPE*

(1) Description: A LINK_TYPE represents a tactically significant contact as reported over Link-11 (in M_SERIES_MSG format). The implementation of the LINK_TYPE type is given in Appendix R, p. 238.

(2) Attributes: These elements refer to the LINK_TYPE's Link number, its relative position from DLRP (Data Link Reference Point), the time of the observation, the TRACK category, the TRACK identity, and its altitude (if air).

(3) Creation Operations: A LINK_TYPE is created by CONVERT_M_SERIES_MSG_TO_LINK_TYPE.

(4) Update Operations: Since the information used to fill an object of LINK_TYPE comes into LCCDS from an external source, LINK_TYPE is not mutable.

f. ABSOLUTE_TIME

(1) Description: ABSOLUTE_TIME represents the year, month, and time of day to the second. The implementation of the ABSOLUTE_TIME type is given in Appendix K, p. 206.

(2) Attributes: YEAR(ABSOLUTE_TIME) refers to the calendar year. MONTH(ABSOLUTE_TIME) refers to the numerical value of the calendar month. DAY(ABSOLUTE_TIME) refers to the calendar day. TIME_OF_DAY(ABSOLUTE_TIME) refers to the number of seconds elapsed in the current day.

(3) Creation Operations: An object of type ABSOLUTE_TIME is created by initiating a function call to MAKE_ABSOLUTE_TIME. Objects of type ABSOLUTE_TIME can also be created though function calls to "+", "-", or NOW.

(4) Update Operations: None.

g. VECTOR_2

(1) Description: Describes a two-dimensional vector defined in terms of floating point numbers, representing a TRACK's course and speed or its bearing and range. The implementation of the VECTOR_2 type is given in Appendix G, p. 186.

(2) Attributes: LENGTH(VECTOR_2) refers to speed or range. DIRECTION(VECTOR_2) refers to course or bearing. X_COORDINATE(VECTOR_2) refers to the X coordinate of the vector. Y_COORDINATE(VECTOR_2) refers to the Y coordinate of the vector.

(3) Creation Operations: Operations that create instances of VECTOR_2 are MAKE_POLAR_VECTOR_2 and MAKE_CARTESIAN_VECTOR_2. Operations that create instances of VECTOR_2 by mathematical manipulations are "+" (the addition of two vectors), "-" (subtraction of one vector from another), DOT_PRODUCT, "*" (multiplication of a vector by a scalar factor).

(4) Update Operations: None.

h. VECTOR_3

(1) Description: Describes a three-dimensional vector defined in terms of floating point numbers. The implementation of the VECTOR_3 type is given in Appendix H, p. 194.

(2) Attributes: Attributes of VECTOR_3 include LENGTH(VECTOR_3), X_COORDINATE(VECTOR_3), Y_COORDINATE(VECTOR_3), Z_COORDINATE(VECTOR_3), THETA(VECTOR_3), and PHI(VECTOR_3).

(3) Creation Operations: Operations that create instances of VECTOR_3 are MAKE_POLAR_VECTOR_3, MAKE_CARTESIAN_VECTOR_3. Operations that create instances of VECTOR_3 by mathematical manipulations are "+" (the addition of two vectors), "-" (subtraction of one vector from another), DOT_PRODUCT, CROSS_PRODUCT, SCALE (multiplication of a vector by a scalar factor)

(4) Update Operations: None.

3. TASK INTEGRATION_SYSTEM:

The purpose of the task is to manage the track database. The task receives data or information from various sources and translate/parse this raw data input into integration system formatted data that the user_interface can graphically display. The task defines entry calls to the various tasks, functions, and procedures that create, delete, or otherwise modify TRACKs and FILTERs. The INTEGRATION_SYSTEM task also provides a timing function for the task PROCESS_LINK_TRACKS that retrieves and modifies Link 11 input. The INTEGRATION_SYSTEM task is necessary to provide a Real_Time environment for the integration system. The task allows parallel processing to take place preventing one function or procedure from dominating the CPU.

A list of the entry calls defined by the task follows:

- Entry CREATE_TRACK: Creates a TRACK and enters it into the TRACK_DATABASE.
- Entry DELETE_TRACK_AND_SEND_TO_HISTORY: Deletes a TRACK from the active TRACK_DATABASE and sends it to history.
- Entry ADD_TRACK_OBSERVATION: Adds an observation to an existing TRACK, using relative position from OWNSHIP as the observation location.
- Entry SET_TRACK_IDENTITY: Sets/changes a TRACK's IDENTITY.
- Entry SET_AMPL_INFO: Sets/changes a TRACK's AMPLIFYING_INFO.
- Entry SET_PLATFORM_CLASS: Sets/changes a TRACK's CLASS.
- Entry SET_VESSEL_NAME: Sets/changes a TRACK's NAME.
- Entry SET_ALTITUDE: Sets/changes a TRACK's ALTITUDE.
- Entry GET_CONTROL: Gets a TRACK's CONTROL.
- Entry SET_CONTROL: Sets/changes a TRACK's CONTROL.

- Entry CHANGE_TRACK_CATEGORY: Sets/changes a TRACK's IDENTITY.
- Entry BUILD_WAYPOINT_SPECIAL_POINT: Builds a WAYPOINT TRACK.
- Entry BUILD_NAV_HAZARD_SPECIAL_POINT: Builds a NAV_HAZARD TRACK.
- Entry BUILD_GENERAL_SPECIAL_POINT: Builds a GENERAL SPECIAL_POINT TRACK.
- Entry BUILD_PATH: Builds a PATH TRACK.
- Entry BUILD_ABSOLUTE_CIRCLE_REGION: Builds an ABSOLUTE CIRCLE REGION TRACK.
- Entry BUILD_RELATIVE_CIRCLE_REGION: Builds a RELATIVE CIRCLE REGION TRACK, with the radius of the circle in yards and position of circle center relative to reference track position.
- Entry BUILD_ABSOLUTE_POLYGON_REGION: Builds an ABSOLUTE POLYGON REGION TRACK.
- Entry BUILD_RELATIVE_POLYGON_REGION: Builds a RELATIVE POLYGON REGION TRACK.
- Entry CHANGE_COURSE: Adds TRACK observation reflecting TRACK's course change.
- Entry CHANGE_SPEED: Adds TRACK observation reflecting TRACK's speed change.
- Entry CHANGE_GLOBAL_POSITION: Adds TRACK observation reflecting TRACK's position change.
- Entry MAKE_DISTANCE_ATOMIC_FILTER: Makes an ATOMIC_FILTER based on distance type attributes and adds it to the current AND_FILTER.

- Entry MAKE_TRACK_CATEGORY_ATOMIC_FILTER: Makes an ATOMIC_FILTER based on TRACK category type attributes and adds it to the current AND_FILTER.
- Entry MAKE_PLATFORM_IDENTITY_ATOMIC_FILTER: Makes an ATOMIC_FILTER based on TRACK identity type attributes and adds it to the current AND_FILTER.
- Entry ADD_AND_FILTER_TO_FILTER: Adds a filled AND_FILTER to the current FILTER.
- Entry CLEAR_FILTER: Clears the FILTER to make way for a new one.
- Entry WRITE_FILTER: Writes a filled FILTER to an archive file for historical purposes.
- Entry FILL_TACPLOT: Fills the tactical display structure with TRACKs that pass FILTER requirements.
- Entry SET_SENSOR_STATUS: Flags the system as to whether or not to accept input from a particular OWNSHIP sensor.
- Entry GET_SENSOR_STATUS: Gets the current input status from a particular OWNSHIP sensor.
- Entry SHUTDOWN: Purges the TRACK_DATABASE, sending each TRACK to an archive file. Also writes archived TRACK info and FILTER info to text files. Aborts the GPS update task.

4. TASK GPS_UPDATE_TASK:

The purpose of the task is to interface to a Global Positioning System via the RS-232 communication port. The task reads in a string of data that represents the geographic position of the ship at the time the data was received, and store the Global Positioning System data in a buffer. The task defines no entry calls but, invokes the procedure Add_Track_Observation which accepts the geographic position reported by the Global Positioning System as a new observation. Retrieves GPS data every four seconds and adds

a new OWNSHIP TRACK observation The task is a separate task because if it were a procedure or function the system would not be released to perform any other operations.

No entry calls are defined from GPS_UPDATE_TASK.

5. TASK LINK_CYCLE:

The purpose of the task is to limit the rate of the Link input. The task has an endless loop that clocks the time period of four seconds between loops. Each loop the task calls the procedure that reads in the Link buffer and processes the M_Series_Messages into Link_Tracks and stores them in the database. The single entry call defined by the task is listed below:

a. entry START_LINK_UPDATE;

Link 11 information request performed every 4 seconds

6. TASK PROCESS_LINK_TRACK:

The purpose of the task is to process the Link 11 M_Series_Messages into Link_Track format. After processing the message buffer the task checks the database to see if the track is active. If the track is found the process updates the track with a Global_Observation. If the track is not found the Track is created and stored in the database.

No entry calls are defined by PROCESS_LINK_TRACK.

B. DATABASE MODEL

Design and development of the database for the LCCDS is driven by four goals:

1. Performance: Does the structure of the database support fast access to the data? Can the system(USER) retrieve and update relevant data within specified response time?

2. Integrity: To what extent does the database guarantee that correct data is stored and is not accidentally corrupted?

3. Understandability: How coherent is the structure of the database to the user? After a long period of time, will it still be understandable to the designers and others?

4. Extensibility: How easily can the database be extended to new applications without disrupting the present or on-going system?

Keeping these goals in mind, we define the requirements/restrictions placed on the database.

1. The object-oriented database is to be implemented in Ada.
2. The database is to be divided into two parts.
   a. An active database in main memory.
   b. A historical database in secondary memory.
3. Develop a Real_Time system.
   a. Time meets the four second Real_Time requirement with respect to start time and completion time of a specific transaction(Task,Procedure,Function).
   b. The current design assumes a single processor system.

Design of the tactical database starts with identification of objects and classes. The initial phase consist of analysis of the objects proposed in reference 35. The requirements are not difficult since most objects are identified by references 34 and 35, but, careful analysis of the objects and their class along with the methods are necessary before starting to build the database. First we establish that the database has only one class the abstract data type Track. Each object of this class has object variables specific to that object

Our objective is to use the object-oriented approach in the databade design. An object-oriented distributed program system is modeled as a collection of task or procedures containing transactions and data objects which synchronize their operations through messages. To elaborate, when discussing Ada tasking and communication complexity for

distributed programs, the key property to be considered is that both consist of a number of processes or task that execute asynchronously in parallel, but communicate and synchronize by message passing.

While considering requirements complexity, looking at distributed programs which realize concurrency by parallel execution of separate tasks and which constrain the concurrency by introducing task communication. We came to the conclusion that program complexity consist of two components:

1. A local complexity which reflects the complexity of the individual task.

2. Communication complexity which reflects the complexity of the interactions among tasks.

A transactions accesses objects indirectly by communication of its desires to the transaction manager, which then sends a message to the appropriate object manager. Although transaction and object managers may maintain more than one transaction or object, we assume, with confidence, that the transaction manager controls on transaction at a time, and each object manager controls one object. The internal structure of a transaction manager consist of two components, the transaction body, and the probe queue. When a transaction request an object, the transaction manager sends a message to the object manager with the request. The object manager either grants or denies the request depending upon whether or not the transaction will create a conflict(deadlock) with some transactions already holding the object. The internal structure of the object manager contains:

1. A LOCK_LIST which holds information about those transactions that currently hold a lock on the object.

2. A REQUEST_LIST which lists those transactions currently having an outstanding request on the object.

3. A COMPATIBILITY_TABLE which holds information on the compatibility of operations on the object.

The compatibility table is used by the concurrency control algorithm. The algorithm is based on the read/write lock model and may allow more than one holder since the object can be shared among transactions requesting read only locks. Concurrency control is insured because we have insisted that all transactions run to completion or they don't start running. We accomplish this by building a schedule of transactions to run. Because task run in parallel, it is important to insure the completion of specific parts of the program or task before allowing the remaining procedures or task to run. By insuring this scheduling holds, the results are the same as if the program or task was running individually.

The database stores the track data which contains all the amplifying information needed to identify the contact. The Identification number is assigned by the integration system at the time the track is stored in the database. Because the data structure is a linked list the track ID numbers can range from one to infinity, with zero reserved for ownship. If the track in local the system will assign the next number to the track, but if the track is a link track the system must check if the track is active or not. If the track is active then the system simply updates the track. However, if the track does not exist the system assigns a system track number and add the numbers to a cross reference tables. The cross reference table is used to keep track of what link track goes with what system track. After the table entry is made the system then stores the track in the database. Each track stored in the database has added to it a link listed which contain each of the global observations. Each global observation contains the global position and time of observation for the specific track. The most recent observation is added to the head of the list enabling the system to retrieve the current position with better time efficiency.

As discussed in the previous chapter, GPS data is received and buffered once every second. The integration system once every four seconds lock the buffer for writing in order to prevent inadvertent changing of data while reading. The integration systems package "navigation handler" reads the GPS data, translate the data to system format as illustrated

in Figure 18 and stores the data as ownship location in the database track zero. Likewise, the LMS 11r the intermediate link 11 processor sends a series of M_Series_Messages through the decoder. The integration system receives the data which is buffered for reading by the integration systems “link processor”. The integration system once every four seconds lock the link buffer to prevent changing of data while reading is taking place. Then read the data and store it in the database by the appropriate track number assigned.

Tracks are stored to secondary memory(History) only when the active track is deleted from the active database. If the system crashes, the active tracks in the active database are lost. However, these tracks can and must be recreated when the system is brought back on line. The user may select any number of tracks from the historical database to review by calling READ_TRACK_FROM_ARCHIVES and entering the track number/numbers desired. The integration system retrieves from secondary memory each track desired and stores a copy of it in an array then passes the array to the user for graphic display.

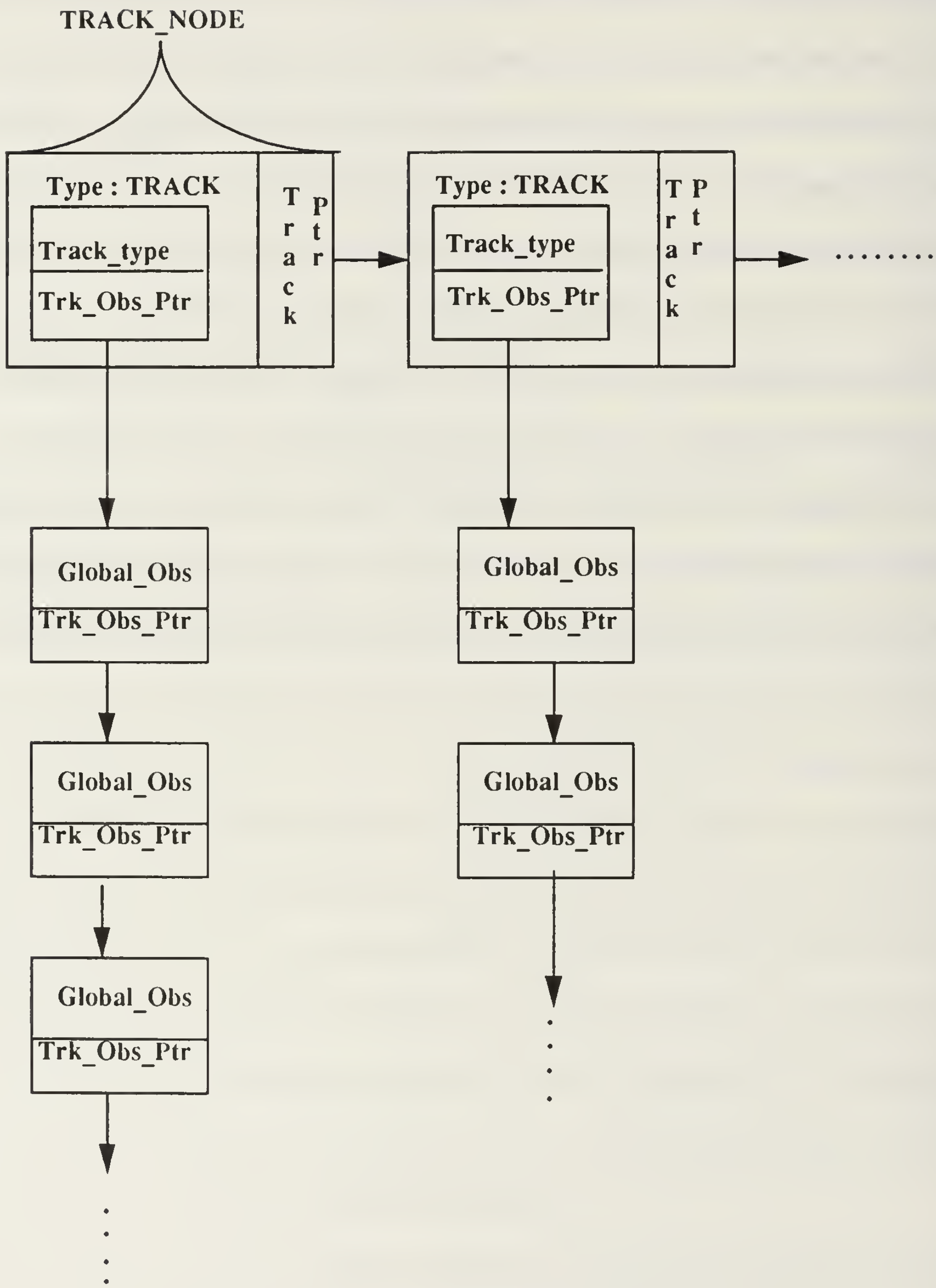


Figure 18 : DATABASE STRUCTURE

C. LINK 11 MODEL

It is important to note that most material related to and involving the link 11 system is classified confidential or higher. This document however, is unclassified, therefore the discussion of the link 11 system and the interface to it is limited to the unclassified portion.

The link 11 signal is transmitted via UHF/HF radio communications to the fleet. We purpose to use an existing system LMS 11r to be an intermediate step between the LCCDS integration system and the link 11 receiver on the platform of choice. The LMS 11r is a unit already tested and in use. The General Specifications and Operational Specifications are, according to our source, in the Department of Defense supply system [Ref. 40].

The Link 11 interface with the integration system consists of a link 11 handler designed inside the integration system and communicating directly with the LMS 11r system. The link handler is an Ada function which breaks a string of characters into the individual parts of the track data type and stores the array of parts in a buffer waiting for the integration system to lock the buffer and read out the data. The integration system then unlocks the buffer and the link handler repeats the process.

Link 11 data consist of two parts: a Data_Link_Reference_Point(DLRP) and a string of tracks reported by fleet assets with reference to the DLRP. The DLRP must be entered manually in the system by the user. The link handler translates the DLRP into a Global_Position and stores it as a regular track. The integration system assigns a special non changing track number to the DLRP that is determined at the time DLRP is entered. This track number will be determined by the system each time DLRP is entered. Utilizing this special track number the system calculates the relative position of the DLRP relative to ownship and the Global_Position of each track in the Link 11 database. The User_Interface selects the reference track and invokes the integration system function Relative_Position to compute the relative position of each Link 11 track to the reference

track. If the user does not select a reference track the system uses ownship as the default reference track and computes the Relative_Position of each Link 11 track relative to ownship.

The track is then stored in the database with a system assigned track number. In order to keep track of which link track matches with which system track, a table is constructed in the integration system. The table contains three elements, the link track number, the corresponding system track number and a pointer to link them together. When an updated set of tracks is received the system searches the table to see if the link track is an active system track. If the link track is found to be an active system track the system updates the Global_Observation of the corresponding system track. If the link track is not found, the system calls create track, assign a track number to the corresponding link track, and stores the track in the database as illustrated in Figure 19.

The integration system scans the link track table for time out every four seconds covering every track in the database designated as link control. The user may at any time take local control of a link track simply by changing the track control to local. A time out event causes the system to drop the link track from the active database. This action is necessary in situations where no updates on the specific track have been received in a pre-assigned time period. By doing so the system removes all inactive link tracks from the active database, freeing up space for new ones. The procedure has no control over local designated tracks. The user must clean house for these user generated tracks or tracks the user has changed from link to local control.

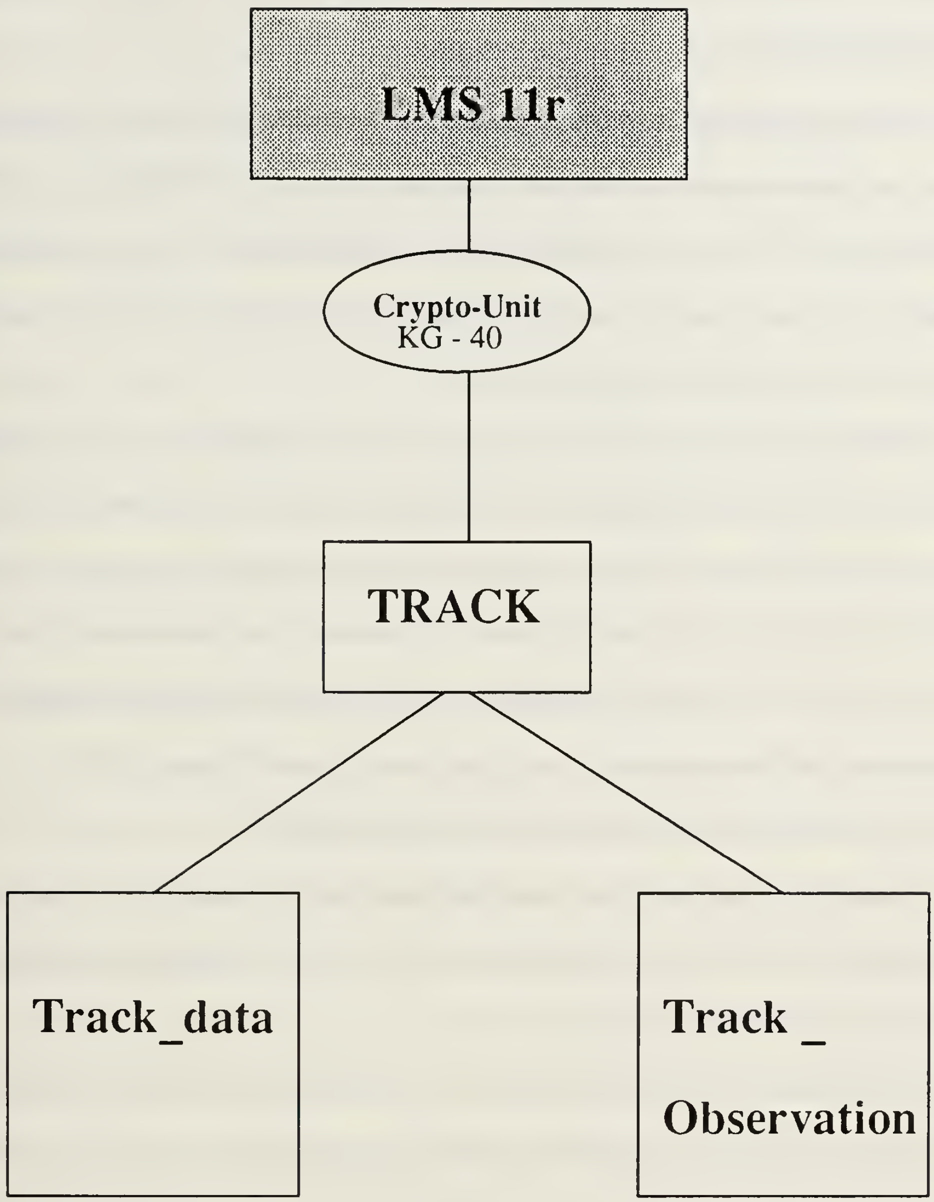


Figure 19 : DATA STRUCTURE DIAGRAM(LINK 11)

V. EVALUATION OF SYSTEM PERFORMANCE

A. FUNCTIONAL

Initial testing of the integration system was conducted by first designing a test program to evaluate each individual requirement [enclosure 1, Ref. 1]. The process of evaluating the integration system included testing for correctness and timing of each procedure, function, and task individually as illustrated in Figures 20 through 26. The test for each individual component was conducted successfully.

The system test program was expanded to test the integration system collectively. To accomplish this testing procedure the integration system was linked to the navigation system for Global Position System data input. Manual tracks were entered as Link tracks to simulate Link 11 input. Each feature of the requirements of enclosure one of Reference 1 was tested for correctness. Timing for a single iteration of the requirements feature was recorded and is illustrated in the timing diagrams Figures 20 through 26.

A list of the test and evaluation of the system follows:

1. Track testing phase: Testing of the Track package required the evaluation of each procedural operation and capability specified by the requirement specification. The list of these steps and their results are:

Allow the user to create a manually input track and store the track in the database: The user may enter a track by either entering the bearing and range to the track from a reference track or by entering a Global_Position of the new track. Timing is well within the Real_Time range and correctness is verified.

The integration system adds a new track to the database when the user manually inputs a track or when a track is received from the link processor is not found in the link to system track number reference table located in the integration system. The integration

system will assign a system track number to the track and store the track in the database. Timing is well within the Real_Time range and correctness is verified.

The user has the option to delete any track from the database simply by identifying the track by the Track_Number, locate and retrieve the track from the database, and call the function DELETE_TRACK. Deletion of a track removes the track from the active database and stores the track and all of the global_observations to history in secondary memory. Timing is well within the Real_Time range and correctness is verified.

The system receives from GPS ownship fix data. Translates the data string into integration system formatted track data and stores the track in the database as track number zero. The system receives from GPS new fix data every second and stores the data in a buffer. The integration system reads the buffer every four seconds and stores the data in the database as the current Global_Observation for track zero. Timing is well within the Real_Time range and correctness is verified.

The user can change the attributes of a track in the database but, cannot change a Global_Observation. The user has the option to record or change the track category and identity or enter any amplifying information about the track. The user can make a manual course and speed change. The integration system will compute and record a new course and speed based on each new Global_Observation received or the manual course and speed entry from the user. When the track location is received as a Global_Position the system will compute the bearing and range to the track from ownship and record the data. Timing is well within the Real_Time range and correctness is verified.

2. Velocity package testing phase: The system determines the velocity of a specified track. Velocity is divided into course and speed of a track Timing is well within the Real_Time range and correctness is verified.

3. Global position package testing phase: The system allows the user to manually input a Global_Position or will automatically convert a Relative_Position to a Global_Position and assign the global position to a specific track. The system assigns a relative position to a specific track from any specified reference track or defaults to ownship as the reference track. Given a Global_Position the system computes the Relative_Position. Timing is well within the Real_Time range and correctness is verified.

4. CPA testing phase: The system determines the closest point of approach between any two specified tracks. The CPA results are true bearing, range, and time of CPA. Timing is well within the Real_Time range and correctness is verified.

5. Filter testing phase: The system can designate a specified filter called an atomic filter and with the mathematical expressions and/or combine a series of these atomic filters into a specific system filter which filters tracks for display only those that meet the specific restrictions placed on the system by the user. Timing is well within the Real_Time range and correctness is verified.

Testing of the integration system takes on two faces. The first is that of a bug or problem finding and removal process. The second is a timing test to see if the individual Functions and/or Procedures meet the Real_Time timing constraints. The timing test is divided into two parts, one to test the complete process run time and the second is testing each iteration of the process. Real Time is defined by NAVSEA as a four second period of time. Testing of the integration system has revealed to date, a safe and comfortable time margin within this Real_Time period in which the system may operate.

While observing the timing graphs in section B this chapter, keep in mind that the times used were generated by the UNIX operating system and rounded off to fit the timing graphs. In each timing case the function tested was the primary function and may include any number

of called functions and procedures. The time considered for each timing graph was the composite time required to execute the primary function. Each iteration of a procedure/ function was also timed.

B. TIMING CHARTS FOR REAL TIME CONSTRAINTS TESTING

Timing and evaluation of the functions and tasks of the integration system was conducted to evaluate the Real_Time requirements for the LCCDS. Each entry in the timing diagrams Figures 20 through 26 correspond to a specific requirement by the sponsor [enclosure 1, Ref. 1]. Each individual entry in the timing diagrams has two timing categories and was conducted as previously discussed. The ***Isolated Module*** category for each entry represents the time required to execute the requirements feature of the procedure, function, or task as a individual unit. The ***System Response*** category for each entry represents the time required to execute the same feature by the integration system. Each entry in the timing diagrams was evaluated against the Real_Time requirement of the four second time period to refresh/fill the TACPLOT for graphic display of the tactical situation.

The integration system is designed such that no single operation will dominate CPU time. The tasks and functions that are executed on a timed cycle require a small amount of the four second time period allowing time for the operations requested by the user. Utilizing these procedures we have developed a set of timing charts that very closely represent the actual CPU time required for the integration system.

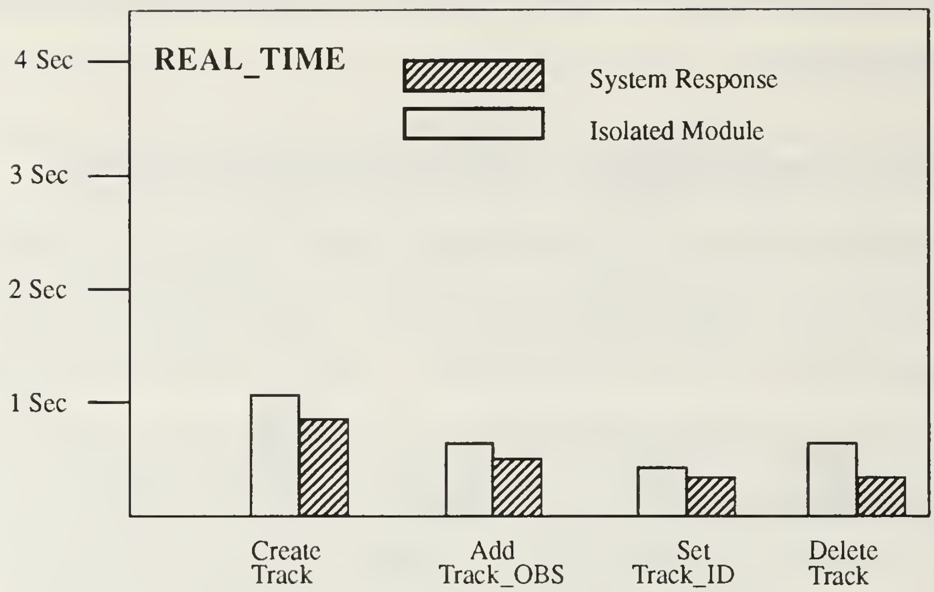


Figure 20 : TIMING DIAGRAM 1

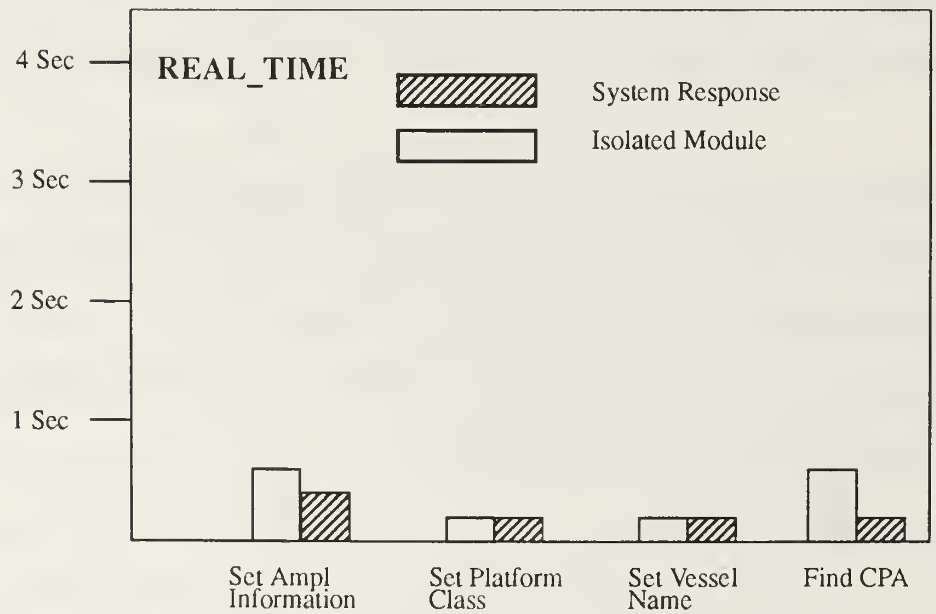


Figure 21 : TIMING DIAGRAM 2

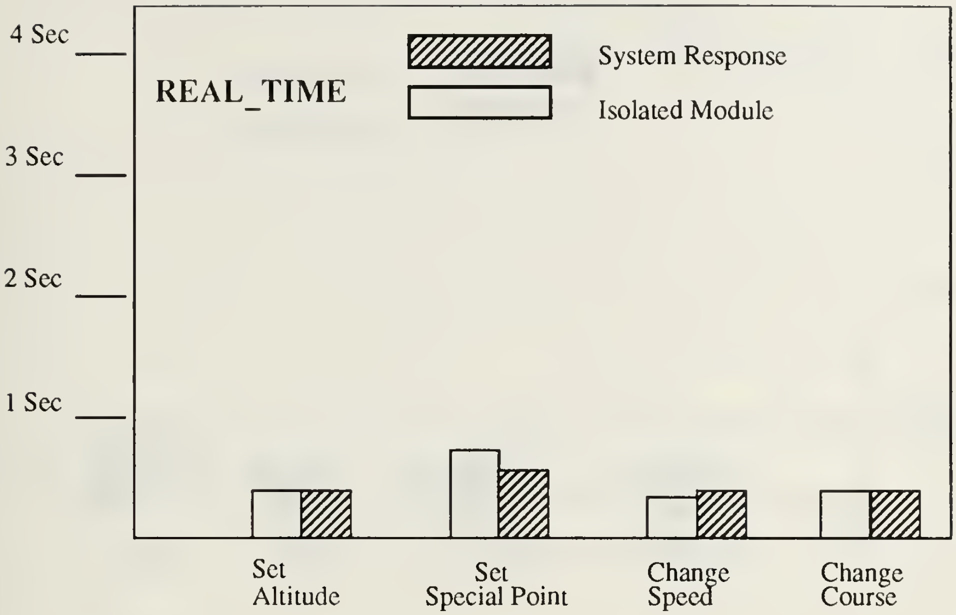


Figure 22 : TIMING DIAGRAM 3

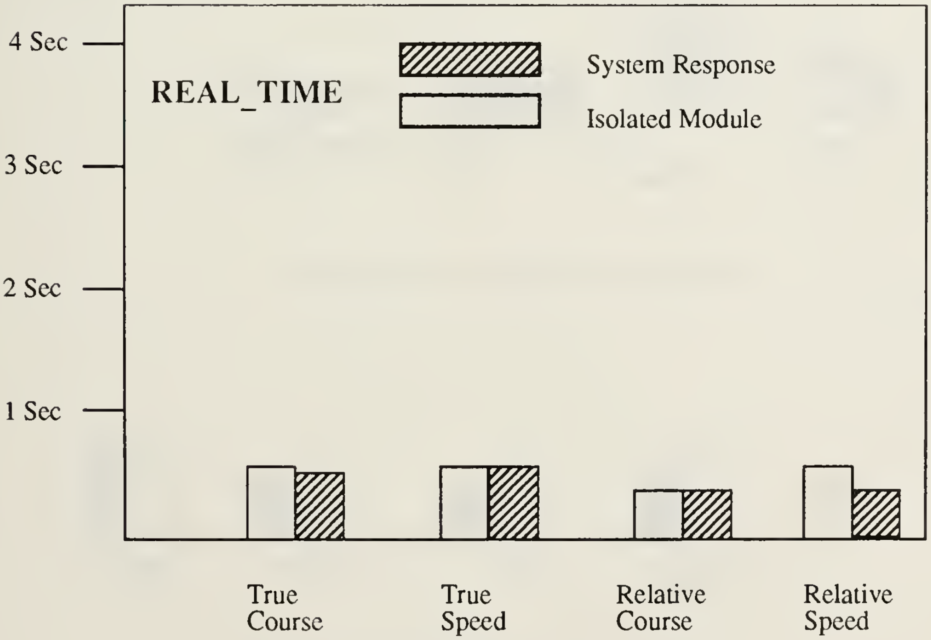


Figure 23 : TIMING DIAGRAM 4

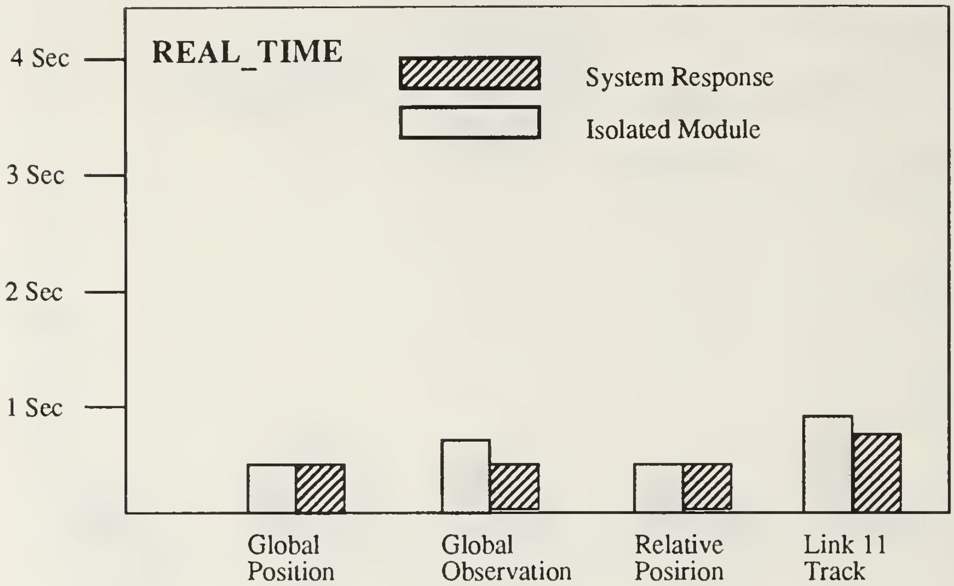


Figure 24 : TIMING DIAGRAM 5

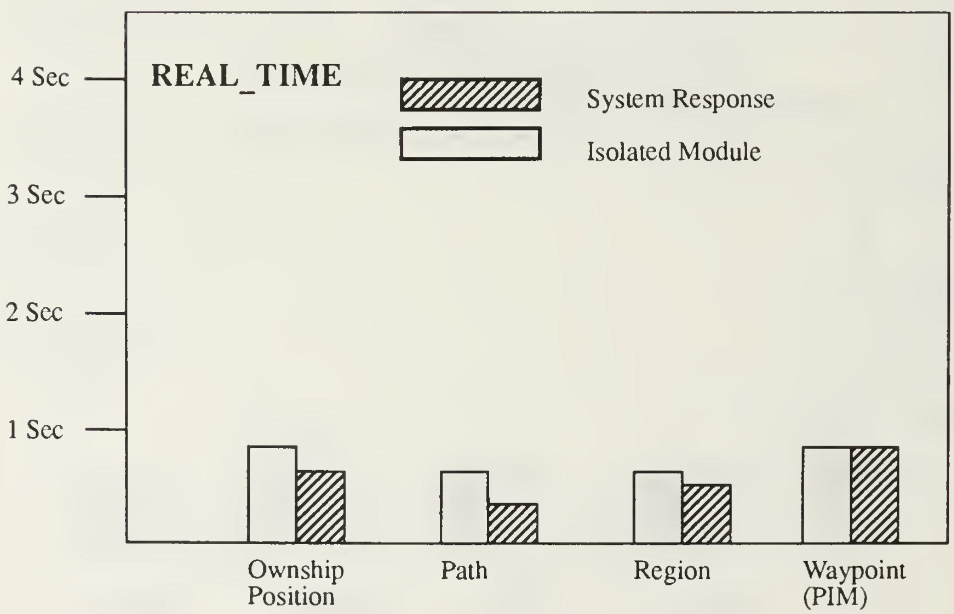


Figure 25 : TIMING DIAGRAM 6

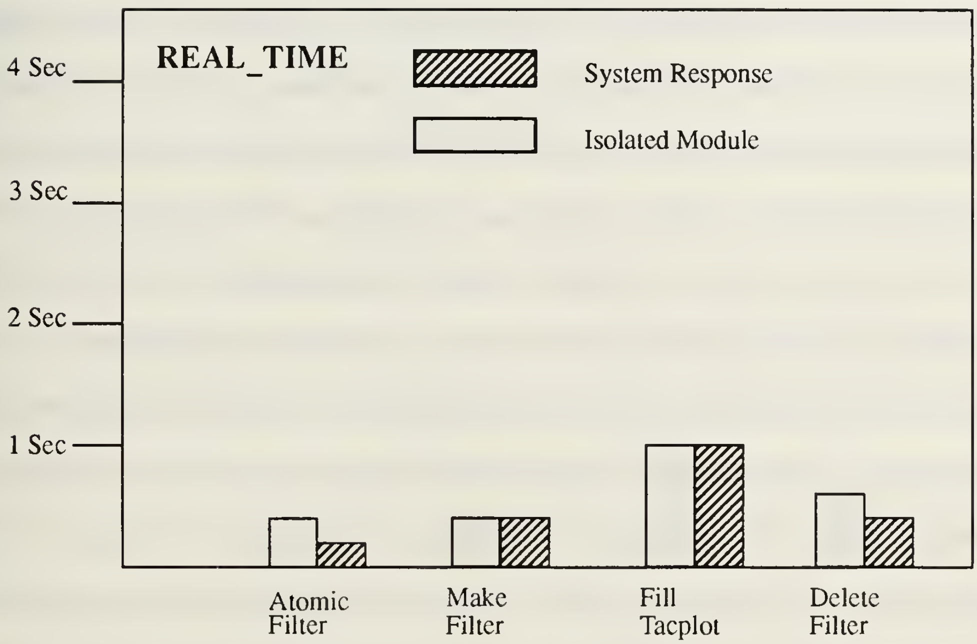


Figure 26 : TIMING DIAGRAM 7

VI. CONCLUSIONS

A. RECOMMENDATIONS

The use of reusable software is an approach that saves time and money. It is a software development technique that works. One of the most serious problems faced today in armed forces acquisition of new systems is the length of time between initial requirements analysis and delivery of a usable system. This generally means that the system delivered is already out of date when it arrives. The LCCDS design, however, takes advantage of rapidly improving commercial computer technology, hardware and software. Specifically, we take advantage of reliable and inexpensive commercial workstation systems. Even more significant is the fact that we can obtain these workstations now vice having to wait for years while someone makes up their mind what the specification for the system should be.

During the period of our research we discovered an interesting fact: there are several different projects being funded to do the same exact thing, to develop a Combat Direction System that can be placed on non NTDS and NTDS ships to assist in the navigation and daily formation steaming functions. A combined effort might produce a workable prototype capable of accomplishing what NAVSEA has mandated. The continuation of the LCCDS will see such a prototype in the fleet and soon after working models.

Considerable effort was expended searching for an existing software unit capable of translating Link 11 data into a format the system could utilize. Our recommendation is to include the LMS- 11r a Logicon product to accomplish this task. The LMS-11r is a unit that is presently in the system and can be obtained with short lead time.

A study designed to research the possibility of incorporating parts of the ATP-1C into the system capabilities would be money well spent. All the necessary elements with the exceptions of the algorithms are built into the system. Adding the required procedures and functions containing the algorithms for computing solutions to ATP-1C requirements and

incorporating them into the system calls would accomplish this requirement. To complete the addition of the ATP-1C solution solver would require the classifying of the system.

B. EVOLUTION OF THE SYSTEM

In order to accommodate the evolutionary changes in the tactical environment resulting from changes in tactics, weapons and sensors, the LCCDS has to be capable of quick and inexpensive software upgrades. This operational flexibility is a paramount requirement. The system must be programmable, to adapt to system failures and the ever changing data structure used in the integration system to support the constant evolution of the system support software. The need for flexibility clearly dictates the use of a general purpose, stored program, commercial computer where parts and upgrades can be accomplished with minimum cost in time and money.

It is necessary to convert the various forms of tactical data from analog to digital representations so that all data in the system can be represented in the same formats. Analog to digital conversion becomes an important hardware priority. The procurement and installation of this hardware must be addressed in the continuation of this body of research. A follow on study would be most appropriate but not necessary as we have completed the initial leg work and have outlined the necessary additions in hardware and software. Speed of conversion and accuracy are the prime objectives in this task. The conversion to digital data representation must be done as close to the source as possible to maintain accuracy throughout the system and resultant data calculations and applications.

Different types of these conversion units must be defined and specific decisions made as to which unit will be used in each of the more specific applications. The on-line analog to digital conversion must be used for vital data sources such as these selected sensors: gyro, pit log and the platforms primary radars.

Automatic radar detection and tracking of targets is another area of research for future projects. This thesis has not explored this vast and complex area. At present some ships of the fleet have a basic manual, rate aided tracking capability for all installed radars. However, on some ships when radar is overwhelmed with tracks, using the conventional grease pencil method of tracking and plotting, the analysis and decision making functions border on hopelessness. There is an obvious need for the auto tracking capability to be installed on all ships of the fleet.

Each ship equipped with LCCDS would have a real time working advantage over ships not equipped with NTDS or the LCCDS. No longer would the commander have to wait for critical data needed to make fast, accurate, life-threatening decisions need for safe ship-handling. The commander would have more confidence in his decisions because of his confidence in the accuracy of the LCCDS system.

APPENDIX A

GUIDE TO DATA TYPES

In order to better understand the integration system this guide to the data types and the location where they can be found is provided:

VECTOR_2_PKG	function	"*"
ABSOLUTE_TIME_PKG	function	"+"
VECTOR_2_PKG	function	"+"
VECTOR_3_PKG	function	"+"
ABSOLUTE_TIME_PKG	function	"-"
VECTOR_2_PKG	function	"-"
VECTOR_3_PKG	function	"-"
ABSOLUTE_TIME_PKG	function	"<"
ABSOLUTE_TIME_PKG	type	ABSOLUTE_TIME
TRACK_PKG	type	ABSOLUTE_VERTEX_ARRAY
TRACK_PKG	function	ABS_CIRCLE_CENTER
TRACK_PKG	function	ABS_REGION_VERTICES
TRACK_PKG	type	ABS_VERTEX_TYPE
TRACK_DATABASE_PKG	function	ACTIVE_TRACK
FILTER_PKG	procedure	ADD_AND_FILTER_TO_FILTER
FILTER_PKG	procedure	ADD_ATOMIC_FILTER_TO_AND_FILTER
TRACK_PKG	procedure	ADD_TRACK_OBSERVATION
TRACK_DATABASE_PKG	procedure	ADD_TRACK_TO_DBASE
TRACK_PKG	subtype	AIR_TRACK_TYPE
TRACK_PKG	function	ALTITUDE
TRACK_PKG	function	AMPL_INFO
TRACK_PKG	package	AMP_STR
FILTER_PKG	type	AND_FILTER
FILTER_PKG	type	AND_FILTER_NODE
FILTER_PKG	type	AND_FILTER_PTR
ANGLE_PKG	subtype	ANGLE
ANGLE_PKG	function	ARCSIN
ANGLE_PKG	function	ARCTAN
FILTER_PKG	type	ATOMIC_FILTER
FILTER_PKG	type	ATOMIC_FILTER_NODE
FILTER_PKG	type	ATOMIC_FILTER_OUT
FILTER_PKG	type	ATOMIC_FILTER_PTR

ANGLE_PKG	subtype	AZIMUTH
RELATIVE_POSITION_PKG	function	BEARING_TO
TRACK_PKG	procedure	BUILD_ABSOLUTE_CIRCLE_REGION
TRACK_PKG	procedure	BUILD_ABSOLUTE_POLYGON_REGION
TRACK_PKG	procedure	BUILD_GENERAL_SPECIAL_POINT
TRACK_PKG	procedure	BUILD_NAV_HAZARD_SPECIAL_POIN
TRACK_PKG	procedure	BUILD_PATH
TRACK_PKG	procedure	BUILD_RELATIVE_CIRCLE_REGION
TRACK_PKG	procedure	BUILD_RELATIVE_POLYGON_REGION
TRACK_PKG	procedure	BUILD_WAYPOINT_SPECIAL_POINT
TRACK_PKG	procedure	CHANGE_COURSE
TRACK_PKG	procedure	CHANGE_GLOBAL_POSITION
TRACK_PKG	procedure	CHANGE_SPEED
TRACK_PKG	procedure	CHANGE_TRACK_CATEGORY
TRACK_PKG	function	CIRCLE_RADIUS
FILTER_PKG	procedure	CLEAR_FILTER
TRACK_PKG	function	CONTROL
TRACK_PKG	type	CONTROL_TYPE
ANGLE_PKG	function	COS
VELOCITY_PKG	function	COURSE
CPA_PKG	type	CPA_TYPE
FILTER_PKG	procedure	CREATE_FILTER_FILE
TRACK_PKG	procedure	CREATE_TRACK
TRACK_PKG	procedure	CREATE_TRACK_FILES
VECTOR_3_PKG	function	CROSS_PRODUCT
TRACK_PKG	function	CURRENT_POSITION
ABSOLUTE_TIME_PKG	function	DAY
ANGLE_PKG	function	DEGREES_TO_RADIANS
TRACK_PKG	procedure	DELETE_TRACK_AND_SEND_TO HISTORY
VECTOR_2_PKG	function	DIRECTION
DISTANCE_PKG	subtype	DISTANCE
FILTER_PKG	type	DISTANCE_ATTRIBUTE_ID
DISTANCE_PKG	function	DISTANCE_IN_NAUTICAL_MILES
VECTOR_2_PKG	function	DOT_PRODUCT
VECTOR_3_PKG	function	DOT_PRODUCT
TRACK_DATABASE_PKG	procedure	DROP_TRACK_FROM_DBASE
GLOBAL_POSITION_PKG	type	EAST_WEST
FILTER_PKG	subtype	EQUALITY_RELATION_ID
FILTER_PKG	function	EVERYTHING
NAVIGATION_PKG	package	E_W_INOUT
FILTER_PKG	type	FILTER

FILTER_PKG	type	FILTER_CATEGORY
FILTER_PKG	package	FILTER_INOUT
CPA_PKG	function	FIND_CPA
GLOBAL_POSITION_PKG	function	FIND_GLOBAL_POSITION
GLOBAL_POSITION_PKG	function	FIND_RELATIVE_POSITION
TRACK_DATABASE_PKG	procedure	FIND_TRACK_IN_DBASE
FILTER_PKG	procedure	FREE_AND_FILTER
FILTER_PKG	procedure	FREE_ATOMIC_FILTER
TRACK_PKG	procedure	FREE_OBS
TRACK_DATABASE_PKG	procedure	FREE_TRK
NAVIGATION_PKG	function	GET_GPS_UPDATE
GLOBAL_POSITION_PKG	procedure	GET_LATITUDE
GLOBAL_POSITION_PKG	procedure	GET_LONGITUDE
SYSTEM_STATUS_PKG	function	GET_STATUS
GLOBAL_POSITION_PKG	type	GLOBAL_POSITION
TRACK_PKG	type	GLOB_OBS_ARRAY
INTEGRATION_SYSTEM_PKG	task	GPS_UPDATE_TASK
GLOBAL_POSITION_PKG	function	GREAT_CIRCLE_BEARING
GLOBAL_POSITION_PKG	function	GREAT_CIRCLE_DISTANCE
RELATIVE_TIME_PKG	function	HOURS
TRACK_PKG	type	IDENTITY_TYPE
INTEGRATION_SYSTEM_PKG	task	INTEGRATION_SYSTEM
VECTOR_2_PKG	function	LENGTH
VECTOR_3_PKG	function	LENGTH
INTEGRATION_SYSTEM_PKG	task	LINK_CYCLE
LINK_PKG	type	LINK_PTR
LINK_PKG	type	LINK_TABLE
LINK_PKG	type	LINK_TYPE
ABSOLUTE_TIME_PKG	function	MAKE_ABSOLUTE_TIME
VECTOR_2_PKG	function	MAKE_CARTESIAN_VECTOR_2
VECTOR_3_PKG	function	MAKE_CARTESIAN_VECTOR_3
FILTER_PKG	procedure	MAKE_DISTANCE_ATOMIC_FILTER
TRACK_PKG	function	MAKE_GLOBAL_OBSERVATION
GLOBAL_POSITION_PKG	function	MAKE_GLOBAL_POSITION
DISTANCE_PKG	function	MAKE_NAUTICAL_MILES_DISTANCE
FILTER_PKG	procedure	MAKE_PLATFORM_IDENTITY_ATOMIC_FILTER
VECTOR_2_PKG	function	MAKE_POLAR_VECTOR_2
VECTOR_3_PKG	function	MAKE_POLAR_VECTOR_3
RELATIVE_TIME_PKG	function	MAKE_RELATIVE_TIME
SPEED_PKG	function	MAKE_SPEED
FILTER_PKG	procedure	MAKE_TRACK_CATEGORY_ATOMIC_FILTER

VELOCITY_PKG	function	MAKE_VELOCITY
TRACK_PKG	subtype	MAN_IN_WATER_TRACK_TYPE
RELATIVE_TIME_PKG	function	MINUTES
ABSOLUTE_TIME_PKG	function	MONTH
TRACK_PKG	function	MOST_RECENT_OBSERVATION
M_SERIES_MSG_PKG	type	M_SERIES_MSG
M_SERIES_MSG_PKG	type	M_SERIES_MSG_BUFFER
TRACK_PKG	subtype	NON_DISPLAYABLE_TRACK_TYPE
VECTOR_2_PKG	function	NORMALIZE
VECTOR_3_PKG	function	NORMALIZE
GLOBAL_POSITION_PKG	type	NORTH_SOUTH
ABSOLUTE_TIME_PKG	function	NOW
TRACK_PKG	subtype	NUM_HISTORY_PTS
TRACK_PKG	subtype	NUM_PATH_PTS
TRACK_PKG	subtype	NUM_VERTICES
NAVIGATION_PKG	package	N_S_INOUT
TRACK_PKG	function	PATH_POINTS
TRACK_PKG	subtype	PATH_TRACK_TYPE
TRACK_PKG	type	PATH_TYPE
VECTOR_3_PKG	function	PHI
TRACK_PKG	function	PLATFORM_CLASS
TRACK_PKG	procedure	PRINT_GLOBAL_POSITION
TRACK_PKG	procedure	PRINT_OBSERVATION_TIME
FILTER_PKG	procedure	PRINT_TIME_OUT is
PROCESS_LINK_TRACKS_PKG	procedure	PROCESS_MSG_BUFFER
TRACK_DATABASE_PKG	procedure	PURGE_ENTIRE_DBASE
ANGLE_PKG	function	RADIANS_TO_DEGREES
RELATIVE_POSITION_PKG	function	RANGE_OF
TRACK_PKG	function	REGION_CATEG
TRACK_PKG	type	REGION_CATEGORY
TRACK_PKG	type	REGION_PLACEMENT
TRACK_PKG	function	REGION_PLCMT
TRACK_PKG	subtype	REGION_TRACK_TYPE
TRACK_PKG	type	REGION_TYPE
FILTER_PKG	type	RELATION_ID
TRACK_PKG	function	RELATIVE_BEARING
TRACK_PKG	function	RELATIVE_CIRCLE_REFERENCE_TRK_NUM
TRACK_PKG	function	RELATIVE_CIRCLE_REFERENCE _TRK_POS
TRACK_PKG	function	RELATIVE_COURSE
RELATIVE_OBSERVATION_PKG	type	RELATIVE_OBSERVATION
RELATIVE_POSITION_PKG	subtype	RELATIVE_POSITION

TRACK_PKG	function	RELATIVE_REGION_REFERENCE_TRK_NUM
TRACK_PKG	function	RELATIVE_REGION_REFERENCE_TRK_POS
TRACK_PKG	function	RELATIVE_SPEED
RELATIVE_TIME_PKG	subtype	RELATIVE_TIME
TRACK_PKG	type	RELATIVE_VERTEX_ARRAY
TRACK_PKG	function	REL_CIRCLE_CENTER
TRACK_PKG	function	REL_REGION_VERTICE
TRACK_PKG	type	REL_VERTEX_TYPE
TRACK_DATABASE_PKG	procedure	RESTORE_ALTERED_TRACK_TO_DATABASE
VECTOR_2_PKG	function	ROTATE
VECTOR_3_PKG	function	SCALE
RELATIVE_TIME_PKG	function	SECONDS
SYSTEM_STATUS_PKG	type	SENSOR
TRACK_PKG	procedure	SET_ALTITUDE
TRACK_PKG	procedure	SET_AMPL_INFO
TRACK_PKG	procedure	SET_CONTROL
TRACK_PKG	procedure	SET_PLATFORM_CLASS
SYSTEM_STATUS_PKG	procedure	SET_STATUS
TRACK_PKG	procedure	SET_TRACK_IDENTITY
TRACK_PKG	procedure	SET_VESSEL_NAME
ANGLE_PKG	function	SIN
VELOCITY_PKG	function	SPD
TRACK_PKG	type	SPECIAL_POINT_CATEGORY
TRACK_PKG	subtype	SPECIAL_POINT_TRACK_TYPE
TRACK_PKG	type	SPECIAL_POINT_TYPE
TRACK_PKG	function	SPEC_POINT_CATEGORY
SPEED_PKG	subtype	SPEED
SPEED_PKG	function	SPEED_IN_KNOTS
VECTOR_2_PKG	function	SQRT
VECTOR_3_PKG	function	SQRT
SYSTEM_STATUS_PKG	type	STATUS
TRACK_PKG	subtype	SUBSURFACE_TRACK_TYPE
TRACK_PKG	subtype	SURFACE_TRACK_TYPE
SYSTEM_STATUS_PKG	type	SYSTEM_STATUS
TRACK_PKG	function	TARGET_RELATIVE_VELOCITY
INTEGRATION_SYSTEM_PKG	package	TC_INOUT
FILTER_PKG	function	TEST_ATOMIC_FILTER
FILTER_PKG	function	TEST_FILTER
VECTOR_3_PKG	function	THETA
ABSOLUTE_TIME_PKG	function	TIME_OF_DAY
TRACK_PKG	type	TRACK

TRACK_PKG	type	TRACK_CATEGORY
TRACK_DATABASE_PKG	type	TRACK_DATABASE
TRACK_PKG	package	TRACK_DATA_OUT
TRACK_PKG	procedure	TRACK_HISTORY
TRACK_PKG	function	TRACK_IDENTITY
TRACK_PKG	function	TRACK_ID_NUMBER
TRACK_DATABASE_PKG	type	TRACK_NODE
TRACK_PKG	type	TRACK_OBS
TRACK_PKG	package	TRACK_OBS_OUT
TRACK_PKG	type	TRACK_OBS_PTR
TRACK_DATABASE_PKG	type	TRACK_PTR
TRACK_PKG	type	TRACK_TYPE
TRACK_PKG	function	TRK_CATEGORY
TRACK_PKG	function	TRUE_BEARING
TRACK_PKG	function	TRUE_COURSE
TRACK_PKG	function	TRUE_SPEED
TRACK_PKG	function	TRUE_VELOCITY
TRACK_PKG	type	T_OBS
TRACK_PKG	procedure	UPDATE_RELATIVE_CIRCLE_REFERENCE_TRK_POS
TRACK_PKG	procedure	UPDATE_RELATIVE_REGION_REFERENCE_TRK_POS
VECTOR_2_PKG	type	VECTOR_2
VECTOR_3_PKG	type	VECTOR_3
VELOCITY_PKG	subtype	VELOCITY
TRACK_PKG	function	VESSEL_NAME
INTEGRATION_SYSTEM_PKG	package	VPKG
TRACK_PKG	package	V_AND_C_STR
TRACK_PKG	function	WAYPNT
TRACK_PKG	type	WAYPOINT_ARRAY
TRACK_PKG	type	WAYPOINT_TYPE
FILTER_PKG	procedure	WRITE_FILTER
FILTER_PKG	procedure	WRITE_FILTER_ARCHIVES_TO_TEXT_FILE
TRACK_PKG	procedure	WRITE_TRACK_ARCHIVES_TO_TEXT_FILE
VECTOR_2_PKG	function	X_COORDINATE
VECTOR_3_PKG	function	X_COORDINATE
ABSOLUTE_TIME_PKG	function	YEAR
VECTOR_2_PKG	function	Y_COORDINATE
VECTOR_3_PKG	function	Y_COORDINATE
VECTOR_3_PKG	function	Z_COORDINATE

APPENDIX B

INTEGRATION SYSTEM

```
--
==========================================================================
--
-- Authors : Richard T. Irwin
--  Willie K. Bolick
--
-- Date : 29 August 1991
--
--
..........................................................................
--
-- Description : Defines tasks INTEGRATION_SYSTEM, GPS_UPDATE_TASK,
LINK_CYCLE
-- and associated entries
--
--
==========================================================================

with TRACK_PKG, GLOBAL_POSITION_PKG, GLOBAL_OBSERVATION_PKG, ANGLE_PKG,
 SPEED_PKG, DISTANCE_PKG, RELATIVE_POSITION_PKG, FILTER_PKG,
TACPLOT_PKG,
 SYSTEM_STATUS_PKG, ABSOLUTE_TIME_PKG;

use TRACK_PKG, GLOBAL_POSITION_PKG, GLOBAL_OBSERVATION_PKG, ANGLE_PKG,
 SPEED_PKG, DISTANCE_PKG, RELATIVE_POSITION_PKG, FILTER_PKG,
TACPLOT_PKG,
 SYSTEM_STATUS_PKG, ABSOLUTE_TIME_PKG;

package INTEGRATION_SYSTEM_PKG is

 -- Contains entries that deal with procedures to alter the main
 -- TRACK_DATABASE, the FILTER, or the SYSTEM_STATUS
 task INTEGRATION_SYSTEM is

 -- Creates a TRACK and enters it into the TRACK_DATABASE
 entry CREATE_TRACK
 ( OBS : in GLOBAL_OBSERVATION;
```

```
TRK_CAT : in TRACK_CATEGORY );

-- Deletes a TRACK from the TRACK_DATABASE and sends it to history
entry DELETE_TRACK_AND_SEND_TO_HISTORY
( TRK_NUM : in NATURAL );

-- Adds an observation to an existing TRACK
entry ADD_TRACK_OBSERVATION
( TRK_NUM : in NATURAL;
OBS : in GLOBAL_OBSERVATION );

-- Adds an observation to an existing TRACK, using relative position
from
-- OWNSHIP as the observation location
entry ADD_TRACK_OBSERVATION
( TRK_NUM : in NATURAL;
POS : in RELATIVE_POSITION );

-- Sets/changes TRACK's IDENTITY
entry SET_TRACK_IDENTITY
( TRK_NUM : in NATURAL;
TID : in IDENTITY_TYPE );

-- Sets/changes TRACK's AMPLIFYING_INFO
entry SET_AMPL_INFO
( TRK_NUM : in NATURAL;
AMP : in STRING );

-- Sets/changes TRACK's CLASS
entry SET_PLATFORM_CLASS
( TRK_NUM : in NATURAL;
PC : in STRING );

-- Sets/changes TRACK's NAME
entry SET_VESSEL_NAME
( TRK_NUM : in NATURAL;
VES : in STRING );

-- Sets/changes TRACK's ALTITUDE
entry SET_ALTITUDE
( TRK_NUM : in NATURAL;
```

```
ALT : in DISTANCE );

-- Gets TRACK's CONTROL
entry GET_CONTROL
( TRK_NUM : in NATURAL;
CON : out CONTROL_TYPE );

-- Sets/changes TRACK's CONTROL
entry SET_CONTROL
( TRK_NUM : in NATURAL;
CON : in CONTROL_TYPE );

-- Sets/changes TRACK's IDENTITY
entry CHANGE_TRACK_CATEGORY
( TRK_NUM : in NATURAL;
CAT : in TRACK_CATEGORY );

-- Builds a WAYPOINT TRACK
entry BUILD_WAYPOINT_SPECIAL_POINT
( POS : in GLOBAL_POSITION;
TYME : in ABSOLUTE_TIME ); -- time to waypoint

-- Builds a NAV_HAZARD TRACK
entry BUILD_NAV_HAZARD_SPECIAL_POINT
( OBS : in GLOBAL_OBSERVATION );

-- Builds a GENERAL SPECIAL_POINT TRACK
entry BUILD_GENERAL_SPECIAL_POINT
( OBS : in GLOBAL_OBSERVATION );

-- Builds a PATH TRACK
entry BUILD_PATH
( PTS : in WAYPOINT_ARRAY ); -- points in path

-- Builds an ABSOLUTE CIRCLE REGION TRACK
entry BUILD_ABSOLUTE_CIRCLE_REGION
( RAD : in DISTANCE; -- radius of circle(yds)
CTR : in GLOBAL_POSITION ); -- posn of circle center

-- Builds a RELATIVE CIRCLE REGION TRACK
entry BUILD_RELATIVE_CIRCLE_REGION
```

```
( RAD : in DISTANCE; -- radius of circle(yds)
CTR : in RELATIVE_POSITION; -- posn of circle center relative
            -- to ref trk pos
REF_TRK_NUM : in NATURAL ); -- reference track number

-- Builds an ABSOLUTE POLYGON REGION TRACK
entry BUILD_ABSOLUTE_POLYGON_REGION
( AVA : in ABSOLUTE_VERTEX_ARRAY ); -- pts in polygon

-- Builds a RELATIVE POLYGON REGION TRACK
entry BUILD_RELATIVE_POLYGON_REGION
( RVA : in RELATIVE_VERTEX_ARRAY; -- pts in poly reltv to ref trk
            -- position
REF_TRK_NUM : in NATURAL ); -- reference track number

-- Adds TRACK observation reflecting TRACK's course change
entry CHANGE_COURSE
( TRK_NUM : in NATURAL;
CRS : in ANGLE );

-- Adds TRACK observation reflecting TRACK's speed change
entry CHANGE_SPEED
( TRK_NUM : in NATURAL;
SPD : in SPEED );

-- Adds TRACK observation reflecting TRACK's position change
entry CHANGE_GLOBAL_POSITION
( TRK_NUM : in NATURAL;
POS : in GLOBAL_POSITION );

-- Makes an ATOMIC_FILTER based on distance type attributes and adds it
-- to the current AND_FILTER
entry MAKE_DISTANCE_ATOMIC_FILTER
( DAF_ATTRIB_ID : in DISTANCE_ATTRIBUTE_ID;
DAF_LIMIT : in DISTANCE;
DAF_REF_TRK_NUM : in NATURAL;
DAF_RELATION : in RELATION_ID );

-- Makes an ATOMIC_FILTER based on TRACK category type attributes and
adds
-- it to the current AND_FILTER
```

```
entry MAKE_TRACK_CATEGORY_ATOMIC_FILTER
( TCAF_DESIRED_TRK_CAT : in TRACK_CATEGORY;
TCAF_EQ_REL_ID : in EQUALITY_RELATION_ID );

-- Makes an ATOMIC_FILTER based on TRACK identity type attributes and
adds
-- it to the current AND_FILTER
entry MAKE_PLATFORM_IDENTITY_ATOMIC_FILTER
( PIAF_DESIRED_PLAT_ID : in IDENTITY_TYPE;
PIAF_EQ_REL_ID : in EQUALITY_RELATION_ID );

-- Adds a filled AND_FILTER to the current FILTER
entry ADD_AND_FILTER_TO_FILTER;

-- Clears the FILTER to make way for a new one
entry CLEAR_FILTER;

-- Writes a filled FILTER to an archive file for historical purposes
entry WRITE_FILTER;

-- Fills the tactical display structure with TRACKs that pass FILTER
-- requirements
entry FILL_TACPLOT;

-- Flags the system as to whether or not to accept input from a
particular
-- OWNSHIP sensor
entry SET_SENSOR_STATUS
( SENSER : in SENSOR;
SENSER_STATUS : in STATUS );

-- Gets the current input status from a particular OWNSHIP sensor
entry GET_SENSOR_STATUS
( SENSER : in SENSOR;
SENSER_STATUS : out STATUS );

-- Purges the TRACK_DATABASE, sending each TRACK to an archive file.
-- Also writes archived TRACK info and FILTER info to text files.
-- Aborts the GPS update task.
entry SHUTDOWN;
```

```
 end INTEGRATION_SYSTEM;

 -- Retrieves GPS data every 4 seconds and adds a new OWNSHIP TRACK
 -- observation
 task GPS_UPDATE_TASK;

 -- Performs timing function for LINK-11 updates
 task LINK_CYCLE is
 entry START_LINK_UPDATE;
 end LINK_CYCLE;

end INTEGRATION_SYSTEM_PKG;

--
========================================================================
--
-- Authors : Richard T. Irwin
--  Willie K. Bolick
--
-- Date : 29 August 1991
--
--
========================================================================

with TRACK_DATABASE_PKG, VECTOR_2_PKG, CALENDAR, NAVIGATION_PKG;

use TRACK_DATABASE_PKG, VECTOR_2_PKG, CALENDAR, NAVIGATION_PKG;

package body INTEGRATION_SYSTEM_PKG is

 package APKG renames TRACK_PKG.AMP_STR;
 package VPKG renames TRACK_PKG.V_AND_C_STR;
 use APKG, VPKG;

 package INTEGER_INOUT is new INTEGER_IO ( INTEGER );
 package TC_INOUT is new ENUMERATION_IO ( TRACK_CATEGORY );
 use TC_INOUT;

 TRACK_DB : TRACK_DATABASE;

 DIST_AT_FILT : ATOMIC_FILTER;
```

```
TRK_CAT_AT_FILT : ATOMIC_FILTER ( TRACK_CATEGORY_FILTER );
PLTFM_ID_AT_FILT : ATOMIC_FILTER ( PLATFORM_IDENTITY_FILTER );
AND_FILTUR : AND_FILTER;
FILTUR : FILTER;

ACTIVE_TRACK : TRACK;
OTHER_TRACK : TRACK;
OWNSHIP : TRACK;

SYSTUM_STATUS : SYSTEM_STATUS;

LAST_TRK_NUM : NATURAL := 0;

VS1 : VPKG.VSTRING;
VS2 : APKG.VSTRING;

OBS : GLOBAL_OBSERVATION;
POS : GLOBAL_POSITION;

TNUM : NATURAL;

PASSED_FILTER : BOOLEAN;

task body INTEGRATION_SYSTEM is

begin

loop

select
--
.............................CREATE_TRACK...............................
accept CREATE_TRACK
( OBS : in GLOBAL_OBSERVATION;
TRK_CAT : in TRACK_CATEGORY ) do

-- Restore previous ACTIVE_TRACK to TRACK_DATABASE before creating
-- new one
RESTORE_ALTERED_TRACK_TO_DATABASE ( ACTIVE_TRACK, TRACK_DB );
```

```
 CREATE_TRACK ( OBS, LAST_TRK_NUM, ACTIVE_TRACK );

-- Default is UNKNOWN, so don't change if UNKNOWN
 if TRK_CAT /= TRACK_PKG.UNKNOWN then
CHANGE_TRACK_CATEGORY ( ACTIVE_TRACK, TRK_CAT );
end if;

ADD_TRACK_TO_DBASE ( ACTIVE_TRACK, TRACK_DB );

 -- Keep OWNSHIP up-to-date
 if TRACK_ID_NUMBER ( ACTIVE_TRACK ) = 0 then
 OWNSHIP := ACTIVE_TRACK;
 end if;

 end;

 or
--
......................DELETE_TRACK_AND_SEND_TO_HISTORY.................
 accept DELETE_TRACK_AND_SEND_TO_HISTORY
 ( TRK_NUM : in NATURAL ) do

 FIND_TRACK_IN_DBASE ( TRK_NUM, ACTIVE_TRACK, TRACK_DB );
 DROP_TRACK_FROM_DBASE ( TRACK_DB );

   -- Set OWNSHIP as the ACTIVE_TRACK following a deletion
 FIND_TRACK_IN_DBASE ( 0, ACTIVE_TRACK, TRACK_DB );

 end;

 or
--
.........................ADD_TRACK_OBSERVATION..........................
 accept ADD_TRACK_OBSERVATION
 ( TRK_NUM : in NATURAL;
 OBS : in GLOBAL_OBSERVATION ) do

 FIND_TRACK_IN_DBASE ( TRK_NUM, ACTIVE_TRACK, TRACK_DB );
 ADD_TRACK_OBSERVATION ( ACTIVE_TRACK, OBS );

 -- Keep OWNSHIP up-to-date
```

```
if TRACK_ID_NUMBER ( ACTIVE_TRACK ) = 0 then
OWNSHIP := ACTIVE_TRACK;
end if;

end;

or
--
.......................ADD_TRACK_OBSERVATION...........................
accept ADD_TRACK_OBSERVATION
( TRK_NUM : in NATURAL;
POS : in RELATIVE_POSITION ) do

FIND_TRACK_IN_DBASE ( TRK_NUM, ACTIVE_TRACK, TRACK_DB );

  -- Convert the RELATIVE_POSITION observation to a GLOBAL_OBSERVATION
OBS := MAKE_GLOBAL_OBSERVATION ( OWNSHIP, ACTIVE_TRACK, POS );

ADD_TRACK_OBSERVATION ( ACTIVE_TRACK, OBS );

end;

or
--
.........................SET_TRACK_IDENTITY............................
accept SET_TRACK_IDENTITY
( TRK_NUM : in NATURAL;
TID : in IDENTITY_TYPE ) do

FIND_TRACK_IN_DBASE ( TRK_NUM, ACTIVE_TRACK, TRACK_DB );
SET_TRACK_IDENTITY ( ACTIVE_TRACK, TID );

end;

or
--
............................SET_AMPL_INFO..............................
accept SET_AMPL_INFO
( TRK_NUM : in NATURAL;
AMP : in STRING ) do

FIND_TRACK_IN_DBASE ( TRK_NUM, ACTIVE_TRACK, TRACK_DB );
```

```
   -- Convert STRING to a VSTRING ( variable STRING )
VS2 := VSTR ( AMP );

SET_AMPL_INFO ( ACTIVE_TRACK, VS2 );

end;

or
--
.........................SET_PLATFORM_CLASS...........................
accept SET_PLATFORM_CLASS
( TRK_NUM : in NATURAL;
PC : in STRING ) do

FIND_TRACK_IN_DBASE ( TRK_NUM, ACTIVE_TRACK, TRACK_DB );

   -- Convert STRING to a VSTRING ( variable STRING )
VS1 := VSTR ( PC );

SET_PLATFORM_CLASS ( ACTIVE_TRACK, VS1 );

end;

or
--
............................SET_VESSEL_NAME...........................
accept SET_VESSEL_NAME
( TRK_NUM : in NATURAL;
VES : in STRING ) do

FIND_TRACK_IN_DBASE ( TRK_NUM, ACTIVE_TRACK, TRACK_DB );

   -- Convert STRING to a VSTRING ( variable STRING )
VS1 := VSTR ( VES );

SET_VESSEL_NAME ( ACTIVE_TRACK, VS1 );

end;

or
```

```
--
...........................SET_ALTITUDE...................................
accept SET_ALTITUDE
( TRK_NUM : in NATURAL;
ALT : in DISTANCE ) do

FIND_TRACK_IN_DBASE ( TRK_NUM, ACTIVE_TRACK, TRACK_DB );
SET_ALTITUDE ( ACTIVE_TRACK, ALT );

end;

or
--
...........................GET_CONTROL....................................
accept GET_CONTROL
( TRK_NUM : in NATURAL;
CON : out CONTROL_TYPE ) do

FIND_TRACK_IN_DBASE ( TRK_NUM, ACTIVE_TRACK, TRACK_DB );
CON := CONTROL ( ACTIVE_TRACK );

end;

or
--
............................SET_CONTROL...................................
accept SET_CONTROL
( TRK_NUM : in NATURAL;
CON : in CONTROL_TYPE ) do

FIND_TRACK_IN_DBASE ( TRK_NUM, ACTIVE_TRACK, TRACK_DB );
SET_CONTROL ( ACTIVE_TRACK, CON );

end;

or
--
.........................CHANGE_TRACK_CATEGORY.........................
accept CHANGE_TRACK_CATEGORY
( TRK_NUM : in NATURAL;
CAT : in TRACK_CATEGORY ) do
```

```
FIND_TRACK_IN_DBASE ( TRK_NUM, ACTIVE_TRACK, TRACK_DB );
CHANGE_TRACK_CATEGORY ( ACTIVE_TRACK, CAT );

end;

or
--
.......................BUILD_WAYPOINT_SPECIAL_POINT...................
accept BUILD_WAYPOINT_SPECIAL_POINT
( POS : in GLOBAL_POSITION;
TYME : in ABSOLUTE_TIME ) do

-- Restore previous ACTIVE_TRACK to TRACK_DATABASE before creating
-- new one
RESTORE_ALTERED_TRACK_TO_DATABASE ( ACTIVE_TRACK, TRACK_DB );

OBS.POSITION := POS;
OBS.OBSERVATION_TIME := TYME;
OBS.COURSE_AND_SPEED := MAKE_CARTESIAN_VECTOR_2 ( 0.0, 0.0 );

CREATE_TRACK ( OBS, LAST_TRK_NUM, OTHER_TRACK );

   -- Changes TRACK_CATEGORY to SPECIAL_POINT, WAYPOINT & fills
   -- waypoint data
BUILD_WAYPOINT_SPECIAL_POINT ( OTHER_TRACK, POS, TYME );

ACTIVE_TRACK := OTHER_TRACK;
ADD_TRACK_TO_DBASE ( ACTIVE_TRACK, TRACK_DB );

end;

or
--
.......................BUILD_NAV_HAZARD_SPECIAL_POINT.................
accept BUILD_NAV_HAZARD_SPECIAL_POINT
( OBS : in GLOBAL_OBSERVATION ) do

-- Restore previous ACTIVE_TRACK to TRACK_DATABASE before creating
-- new one
RESTORE_ALTERED_TRACK_TO_DATABASE ( ACTIVE_TRACK, TRACK_DB );
```

```
CREATE_TRACK ( OBS, LAST_TRK_NUM, OTHER_TRACK );

  -- Changes TRACK_CATEGORY to SPECIAL_POINT, NAV_HAZARD & fills
  -- nav_hazard data
BUILD_NAV_HAZARD_SPECIAL_POINT ( OTHER_TRACK );

ACTIVE_TRACK := OTHER_TRACK;
ADD_TRACK_TO_DBASE ( ACTIVE_TRACK, TRACK_DB );

end;

or
--
.........................BUILD_GENERAL_SPECIAL_POINT....................
accept BUILD_GENERAL_SPECIAL_POINT
( OBS : in GLOBAL_OBSERVATION ) do

-- Restore previous ACTIVE_TRACK to TRACK_DATABASE before creating
-- new one
RESTORE_ALTERED_TRACK_TO_DATABASE ( ACTIVE_TRACK, TRACK_DB );

CREATE_TRACK ( OBS, LAST_TRK_NUM, OTHER_TRACK );

  -- Changes TRACK_CATEGORY to SPECIAL_POINT, GENERAL
BUILD_GENERAL_SPECIAL_POINT ( OTHER_TRACK );

ACTIVE_TRACK := OTHER_TRACK;
ADD_TRACK_TO_DBASE ( ACTIVE_TRACK, TRACK_DB );

end;

or
--
................................BUILD_PATH..............................
accept BUILD_PATH
( PTS : in WAYPOINT_ARRAY ) do

-- Restore previous ACTIVE_TRACK to TRACK_DATABASE before creating
-- new one
RESTORE_ALTERED_TRACK_TO_DATABASE ( ACTIVE_TRACK, TRACK_DB );
```

```
   -- Use 1st path waypoint as last observation's position
OBS.POSITION := PTS ( 0 ).POSITION;
OBS.OBSERVATION_TIME := NOW;
OBS.COURSE_AND_SPEED := MAKE_CARTESIAN_VECTOR_2 ( 0.0, 0.0 );

CREATE_TRACK ( OBS, LAST_TRK_NUM, OTHER_TRACK );

   -- Changes TRACK_CATEGORY to PATH & fills points
BUILD_PATH ( OTHER_TRACK, PTS );

ACTIVE_TRACK := OTHER_TRACK;
ADD_TRACK_TO_DBASE ( ACTIVE_TRACK, TRACK_DB );

end;

or
--
.......................BUILD_ABSOLUTE_CIRCLE_REGION....................
accept BUILD_ABSOLUTE_CIRCLE_REGION
( RAD : in DISTANCE;
CTR : in GLOBAL_POSITION ) do

-- Restore previous ACTIVE_TRACK to TRACK_DATABASE before creating
-- new one
RESTORE_ALTERED_TRACK_TO_DATABASE ( ACTIVE_TRACK, TRACK_DB );

   -- Use circle center position as last observation's position
OBS.POSITION := CTR;
OBS.OBSERVATION_TIME := NOW;
OBS.COURSE_AND_SPEED := MAKE_CARTESIAN_VECTOR_2 ( 0.0, 0.0 );

CREATE_TRACK ( OBS, LAST_TRK_NUM, OTHER_TRACK );

   -- Changes TRACK_CATEGORY to REGION, CIRCLE, ABSOLUTE & fills
   -- circle data
BUILD_ABSOLUTE_CIRCLE_REGION ( OTHER_TRACK, RAD, CTR );

ACTIVE_TRACK := OTHER_TRACK;
ADD_TRACK_TO_DBASE ( ACTIVE_TRACK, TRACK_DB );

end;
```

```
 or
--
........................BUILD_RELATIVE_CIRCLE_REGION...................
 accept BUILD_RELATIVE_CIRCLE_REGION
 ( RAD : in DISTANCE;
 CTR : in RELATIVE_POSITION;
 REF_TRK_NUM : in NATURAL ) do

   -- Get region's reference TRACK's position
 FIND_TRACK_IN_DBASE ( REF_TRK_NUM, ACTIVE_TRACK, TRACK_DB );
 OBS := MOST_RECENT_OBSERVATION ( ACTIVE_TRACK );

 CREATE_TRACK ( OBS, LAST_TRK_NUM, OTHER_TRACK );

   -- Changes TRACK_CATEGORY to REGION, CIRCLE, RELATIVE & fills
   -- circle data
 BUILD_RELATIVE_CIRCLE_REGION ( OTHER_TRACK, RAD, CTR, REF_TRK_NUM );

 ACTIVE_TRACK := OTHER_TRACK;
 ADD_TRACK_TO_DBASE ( ACTIVE_TRACK, TRACK_DB );

 end;

 or
--
........................BUILD_ABSOLUTE_POLYGON_REGION...................
 accept BUILD_ABSOLUTE_POLYGON_REGION
 ( AVA : in ABSOLUTE_VERTEX_ARRAY ) do

 -- Restore previous ACTIVE_TRACK to TRACK_DATABASE before creating
 -- new one
 RESTORE_ALTERED_TRACK_TO_DATABASE ( ACTIVE_TRACK, TRACK_DB );

 -- Use 1st polygon point as last observation's position
 OBS.POSITION := AVA ( 0 );
 OBS.OBSERVATION_TIME := NOW;
 OBS.COURSE_AND_SPEED := MAKE_CARTESIAN_VECTOR_2 ( 0.0, 0.0 );

 CREATE_TRACK ( OBS, LAST_TRK_NUM, OTHER_TRACK );
```

```
   -- Changes TRACK_CATEGORY to REGION, POLYGON, ABSOLUTE & fills
   -- vertex points
 BUILD_ABSOLUTE_POLYGON_REGION ( OTHER_TRACK, AVA );

 ACTIVE_TRACK := OTHER_TRACK;
 ADD_TRACK_TO_DBASE ( ACTIVE_TRACK, TRACK_DB );

 end;

 or
--
.......................BUILD_RELATIVE_POLYGON_REGION..................
 accept BUILD_RELATIVE_POLYGON_REGION
 ( RVA : in RELATIVE_VERTEX_ARRAY;
 REF_TRK_NUM : in NATURAL ) do

   -- Get region's reference TRACK's position
 FIND_TRACK_IN_DBASE ( REF_TRK_NUM, ACTIVE_TRACK, TRACK_DB );
 OBS := MOST_RECENT_OBSERVATION ( ACTIVE_TRACK );

 CREATE_TRACK ( OBS, LAST_TRK_NUM, OTHER_TRACK );

   -- Changes TRACK_CATEGORY to REGION, POLYGON, RELATIVE & fills
   -- vertex points
 BUILD_RELATIVE_POLYGON_REGION ( OTHER_TRACK, RVA, REF_TRK_NUM );

 ACTIVE_TRACK := OTHER_TRACK;
 ADD_TRACK_TO_DBASE ( ACTIVE_TRACK, TRACK_DB );

 end;

 or
--
..............................CHANGE_COURSE...........................
 accept CHANGE_COURSE
 ( TRK_NUM : in NATURAL;
 CRS : in ANGLE ) do

 FIND_TRACK_IN_DBASE ( TRK_NUM, ACTIVE_TRACK, TRACK_DB );
 CHANGE_COURSE ( ACTIVE_TRACK, CRS );
```

```
end;

or
--
..............................CHANGE_SPEED.............................
accept CHANGE_SPEED
( TRK_NUM : in NATURAL;
SPD : in SPEED ) do

FIND_TRACK_IN_DBASE ( TRK_NUM, ACTIVE_TRACK, TRACK_DB );
CHANGE_SPEED ( ACTIVE_TRACK, SPD );

end;

or
--
..........................CHANGE_GLOBAL_POSITION.......................
accept CHANGE_GLOBAL_POSITION
( TRK_NUM : in NATURAL;
POS : in GLOBAL_POSITION ) do

FIND_TRACK_IN_DBASE ( TRK_NUM, ACTIVE_TRACK, TRACK_DB );
CHANGE_GLOBAL_POSITION ( ACTIVE_TRACK, POS );

end;

or
--
.....................MAKE_DISTANCE_ATOMIC_FILTER.......................
accept MAKE_DISTANCE_ATOMIC_FILTER
( DAF_ATTRIB_ID : in DISTANCE_ATTRIBUTE_ID;
DAF_LIMIT : in DISTANCE;
DAF_REF_TRK_NUM : in NATURAL;
DAF_RELATION : in RELATION_ID ) do

  -- Find reference TRACK in database
FIND_TRACK_IN_DBASE ( DAF_REF_TRK_NUM, ACTIVE_TRACK, TRACK_DB );

MAKE_DISTANCE_ATOMIC_FILTER ( DAF_ATTRIB_ID, DAF_LIMIT,
ACTIVE_TRACK, DAF_RELATION, DIST_AT_FILT );

ADD_ATOMIC_FILTER_TO_AND_FILTER ( DIST_AT_FILT, AND_FILTUR );
```

```
end;

or --
.....................MAKE_TRACK_CATEGORY_ATOMIC_FILTER.................
accept MAKE_TRACK_CATEGORY_ATOMIC_FILTER
( TCAF_DESIRED_TRK_CAT : in TRACK_CATEGORY;
TCAF_EQ_REL_ID : in EQUALITY_RELATION_ID ) do

MAKE_TRACK_CATEGORY_ATOMIC_FILTER ( TCAF_DESIRED_TRK_CAT,
            TCAF_EQ_REL_ID, TRK_CAT_AT_FILT );

ADD_ATOMIC_FILTER_TO_AND_FILTER ( TRK_CAT_AT_FILT, AND_FILTUR );

end;

or
--
..................MAKE_PLATFORM_IDENTITY_ATOMIC_FILTER.................
accept MAKE_PLATFORM_IDENTITY_ATOMIC_FILTER
( PIAF_DESIRED_PLAT_ID : in IDENTITY_TYPE;
PIAF_EQ_REL_ID : in EQUALITY_RELATION_ID ) do

MAKE_PLATFORM_IDENTITY_ATOMIC_FILTER ( PIAF_DESIRED_PLAT_ID,
            PIAF_EQ_REL_ID, PLTFM_ID_AT_FILT );

ADD_ATOMIC_FILTER_TO_AND_FILTER ( PLTFM_ID_AT_FILT, AND_FILTUR );

end;

or
--
...........................ADD_AND_FILTER_TO_FILTER...................
accept ADD_AND_FILTER_TO_FILTER do

ADD_AND_FILTER_TO_FILTER ( AND_FILTUR, FILTUR );

end;

or
--
.................................CLEAR_FILTER........................
```

```
accept CLEAR_FILTER do

CLEAR_FILTER ( FILTUR );

end;

or
--
.................................WRITE_FILTER............................
accept WRITE_FILTER do

WRITE_FILTER ( FILTUR );

end;

or
--
.................................FILL_TACPLOT............................
accept FILL_TACPLOT do

  EMPTY_TACPLOT;

-- For all TRACKS in the database
for I in 0 .. LAST_TRK_NUM loop

FIND_TRACK_IN_DBASE ( I, ACTIVE_TRACK, TRACK_DB );

  -- If TRACK is found
if TRACK_DATABASE_PKG.ACTIVE_TRACK ( TRACK_DB ) then

   -- Things get tricky when the TRACK is a RELATIVE REGION.
   -- We need to retrieve the reference TRACK's position to
   -- calculate the REGION's current position
if TRK_CATEGORY ( ACTIVE_TRACK ) = REGION then

if REGION_PLCMT ( ACTIVE_TRACK ) = RELATIVE_TO_TRACK then

    -- Store the REGION's track number
TNUM := TRACK_ID_NUMBER ( ACTIVE_TRACK );

if REGION_CATEG ( ACTIVE_TRACK ) = CIRCLE then
```

```
     -- Find the reference's position
FIND_TRACK_IN_DBASE ( RELATIVE_CIRCLE_REFERENCE_TRK_NUM
          ( ACTIVE_TRACK ), ACTIVE_TRACK, TRACK_DB );
POS := CURRENT_POSITION ( ACTIVE_TRACK );
OBS := MOST_RECENT_OBSERVATION ( ACTIVE_TRACK );

     -- Make the REGION the ACTIVE_TRACK again
FIND_TRACK_IN_DBASE ( TNUM, ACTIVE_TRACK, TRACK_DB );

     -- Update the REGION's reference position
UPDATE_RELATIVE_CIRCLE_REFERENCE_TRK_POS ( ACTIVE_TRACK,
               POS );
else -- RELATIVE POLYGON

     -- Find the reference's position
FIND_TRACK_IN_DBASE ( RELATIVE_REGION_REFERENCE_TRK_NUM
          ( ACTIVE_TRACK ), ACTIVE_TRACK, TRACK_DB );
POS := CURRENT_POSITION ( ACTIVE_TRACK );
OBS := MOST_RECENT_OBSERVATION ( ACTIVE_TRACK );

     -- Make the REGION the ACTIVE_TRACK again
FIND_TRACK_IN_DBASE ( TNUM, ACTIVE_TRACK, TRACK_DB );

     -- Update the REGION's reference position
UPDATE_RELATIVE_REGION_REFERENCE_TRK_POS ( ACTIVE_TRACK,
               POS );
end if;

     -- If the RELATIVE REGION's course and speed don't match
     -- the reference's, add an observation
if MOST_RECENT_OBSERVATION ( ACTIVE_TRACK ) /= OBS then
ADD_TRACK_OBSERVATION ( ACTIVE_TRACK, OBS );
end if;

end if;

end if;

    -- Test the TRACK against the FILTER
PASSED_FILTER := TEST_FILTER ( FILTUR, ACTIVE_TRACK );
```

```
    -- If TRACK passes FILTER, add it to TACPLOT
    if PASSED_FILTER then
ADD_TACPLOT_ELEMENT ( ACTIVE_TRACK );
    end if;

end if;

end loop;

end;

or

accept SET_SENSOR_STATUS
( SENSER : in SENSOR;
SENSER_STATUS : in STATUS ) do

SET_STATUS ( SYSTUM_STATUS, SENSER, SENSER_STATUS );

end;

or

accept GET_SENSOR_STATUS
( SENSER : in SENSOR;
SENSER_STATUS : out STATUS ) do

SENSER_STATUS := GET_STATUS ( SYSTUM_STATUS, SENSER );

end;

or

accept SHUTDOWN do

PURGE_ENTIRE_DBASE ( TRACK_DB );
  WRITE_TRACK_ARCHIVES_TO_TEXT_FILE;
  WRITE_FILTER_ARCHIVES_TO_TEXT_FILE;
  abort GPS_UPDATE_TASK;
```

```
end;

end select;

end loop;

end INTEGRATION_SYSTEM;

--
...........................GPS_UPDATE_TASK...........................
task body GPS_UPDATE_TASK is

SECONDS : constant DURATION := 1.0;

-- Update required every 4 seconds
INTERVAL : constant DURATION := 4 * SECONDS;

NEXT_GPS_UPDATE : CALENDAR.TIME := CALENDAR.CLOCK + INTERVAL;
OBS : GLOBAL_OBSERVATION;
SENSER_STATUS : STATUS;

begin

loop

delay DURATION ( NEXT_GPS_UPDATE - CALENDAR.CLOCK );
INTEGRATION_SYSTEM.GET_SENSOR_STATUS ( GPS, SENSER_STATUS );

if SENSER_STATUS = UP then

 -- Get OWNSHIP's position from GPS
OBS := GET_GPS_UPDATE;

INTEGRATION_SYSTEM.ADD_TRACK_OBSERVATION ( 0, OBS );
end if;

NEXT_GPS_UPDATE := NEXT_GPS_UPDATE + INTERVAL;

end loop;

exception
```

```
 when STATUS_ERROR | CONSTRAINT_ERROR =>
 SET_STATUS ( SYSTUM_STATUS, GPS, DOWN );

 end GPS_UPDATE_TASK;

--
..............................LINK_CYCLE...................................
......
 task body LINK_CYCLE is

 SECONDS : constant DURATION := 1.0;

 -- Update required every 4 seconds
 INTERVAL : constant DURATION := 4 * SECONDS;
 NEXT_LINK_UPDATE : CALENDAR.TIME := CALENDAR.CLOCK + INTERVAL;

 begin

 loop

 accept START_LINK_UPDATE;
 NEXT_LINK_UPDATE := NEXT_LINK_UPDATE + INTERVAL;
 delay DURATION ( NEXT_LINK_UPDATE - CALENDAR.CLOCK );

 end loop;

 end LINK_CYCLE;

--
.......................................................................

begin
 null;
end INTEGRATION_SYSTEM_PKG;
```

APPENDIX C

TRACK PACKAGE

```
--
======================================================================
--
-- Authors : Richard T. Irwin
--  Willie K. Bolick
--
-- Date : 29 August 1991
--
--
......................................................................
--
-- Description : Defines abstract data type TRACK and associated
--  functions and procedures
--
--
======================================================================

with ANGLE_PKG, SPEED_PKG, DISTANCE_PKG, VELOCITY_PKG,
ABSOLUTE_TIME_PKG,
 GLOBAL_POSITION_PKG, GLOBAL_OBSERVATION_PKG, VSTRINGS,
 RELATIVE_POSITION_PKG, DIRECT_IO;

use ANGLE_PKG, SPEED_PKG, DISTANCE_PKG, VELOCITY_PKG, ABSOLUTE_TIME_PKG,
 GLOBAL_POSITION_PKG, GLOBAL_OBSERVATION_PKG, RELATIVE_POSITION_PKG;

package TRACK_PKG is

 -- Longest length of AMPL_INFO
 AMP_LEN : constant INTEGER := 80;

 -- Longest length of V_NAME & S_CLASS/A_CLASS
 VES_AND_CLASS_LEN : constant INTEGER := 80;

 -- Maximum allowable points in a path
 MAX_PTS_IN_PATH : constant NATURAL := 50;
```

```
-- Maximum number of history points of a TRACK to be displayed to the
user
MAX_HISTORY_PTS : constant NATURAL := 500;

-- Maximum allowable number of vertices in a polygon REGION TRACK
MAX_VERTICES_IN_POLYGON : constant NATURAL := 20;

subtype NUM_PATH_PTS is
NATURAL range 0 .. MAX_PTS_IN_PATH;

subtype NUM_HISTORY_PTS is
NATURAL range 0 .. MAX_HISTORY_PTS;

subtype NUM_VERTICES is
NATURAL range 0 .. MAX_VERTICES_IN_POLYGON;

-- TRACK history points
type GLOB_OBS_ARRAY is
array ( NUM_HISTORY_PTS range <> ) of GLOBAL_OBSERVATION;

type WAYPOINT_TYPE is
record
POSITION : GLOBAL_POSITION;-- Position of waypoint
TIME_TO : ABSOLUTE_TIME;-- Time to arrive at waypoint
end record;

type WAYPOINT_ARRAY is
array ( NUM_PATH_PTS range <> ) of WAYPOINT_TYPE;

type RELATIVE_VERTEX_ARRAY is
array ( NUM_VERTICES range <> ) of RELATIVE_POSITION;

type ABSOLUTE_VERTEX_ARRAY is
array ( NUM_VERTICES range <> ) of GLOBAL_POSITION;

type TRACK is private;

type TRACK_CATEGORY is
( UNKNOWN, SURFACE_PLATFORM, SUBSURFACE_PLATFORM, AIR_PLATFORM,
REGION, PATH, SPECIAL_POINT, MAN_IN_WATER, NON_DISPLAYABLE );
```

```
type IDENTITY_TYPE is
( FRIENDLY, HOSTILE, NEUTRAL, UNKNOWN );

type CONTROL_TYPE is
( LINK, LOCAL );

type SPECIAL_POINT_CATEGORY is
( GENERAL, WAYPOINT, NAV_HAZARD );

type REGION_CATEGORY is
( CIRCLE, POLYGON );

type REGION_PLACEMENT is
( ABSOLUTE, RELATIVE_TO_TRACK );

package AMP_STR is new VSTRINGS ( AMP_LEN );
use AMP_STR;
package V_AND_C_STR is new VSTRINGS ( VES_AND_CLASS_LEN );
use V_AND_C_STR;

-- Creates a TRACK with its first observation
procedure CREATE_TRACK
( GO : in GLOBAL_OBSERVATION;
LAST_TRACK_ID : in out NATURAL;
TRK : out TRACK );

-- Deletes TRACK and sends its TRACK_TYPE data, as well as its
-- GLOBAL_OBSERVATIONs to secondary storage
procedure DELETE_TRACK_AND_SEND_TO_HISTORY
( TRK : in out TRACK );

-- Creates TRK_FILE & OBS_FILE
procedure CREATE_TRACK_FILES;

-- Retrieves archived TRACK info from secondary storage. Reformats it
into
-- a human readable format and writes it to a secondary storage text
file.
procedure WRITE_TRACK_ARCHIVES_TO_TEXT_FILE;

-- Adds an observation of a TRACK to an existing TRACK object
```

```
procedure ADD_TRACK_OBSERVATION
( TRK : in out TRACK;
GO : in GLOBAL_OBSERVATION );

-- Changes/sets TRACK's IDENTITY_TYPE
procedure SET_TRACK_IDENTITY
( TRK : in out TRACK;
TID : in IDENTITY_TYPE );

-- Changes/sets TRACK's AMPLIFYING_INFO
procedure SET_AMPL_INFO
( TRK : out TRACK;
AMP : in AMP_STR.VSTRING );

-- Changes/sets TRACK's CLASS
procedure SET_PLATFORM_CLASS
( TRK : in out TRACK;
PC : in V_AND_C_STR.VSTRING );-- Class name

-- Changes/sets TRACK's VESSEL NAME
procedure SET_VESSEL_NAME
( TRK : in out TRACK;
VES : in V_AND_C_STR.VSTRING );-- Vessel name

-- Changes/sets TRACK's ALTITUDE
procedure SET_ALTITUDE
( TRK : in out TRACK;
ALT : in DISTANCE );-- Altitude in yards

-- Changes/sets TRACK's CONTROL_TYPE
procedure SET_CONTROL
( TRK : out TRACK;
CON : in CONTROL_TYPE );-- LINK/LOCAL control

-- Changes TRACK's TRACK_CATEGORY
procedure CHANGE_TRACK_CATEGORY
( TRK1 : in out TRACK;
CAT : in TRACK_CATEGORY );

-- Builds a WAYPOINT
procedure BUILD_WAYPOINT_SPECIAL_POINT
```

```
( TRK : in out TRACK;
POS : in GLOBAL_POSITION;
TYME : in ABSOLUTE_TIME );-- Time to arrive at waypoint

-- Builds a NAV_HAZARD
procedure BUILD_NAV_HAZARD_SPECIAL_POINT
( TRK : in out TRACK );

-- Builds a GENERAL SPECIAL_POINT
procedure BUILD_GENERAL_SPECIAL_POINT
( TRK : in out TRACK );

-- Builds a PATH
procedure BUILD_PATH
( TRK : in out TRACK;
PTS : in WAYPOINT_ARRAY );-- Points on the path

-- Builds an ABSOLUTE CIRCLE REGION whose center is an absolute
-- GLOBAL_POSITION
procedure BUILD_ABSOLUTE_CIRCLE_REGION
( TRK : in out TRACK;
RAD : in DISTANCE;-- Radius of circle
CTR : in GLOBAL_POSITION );-- Center of circle

-- Builds a CIRCLE REGION whose center is relative to a reference TRACK
procedure BUILD_RELATIVE_CIRCLE_REGION
( TRK : in out TRACK;
RAD : in DISTANCE;-- Radius of circle
CTR : in RELATIVE_POSITION;-- Center of circle relative
             -- to reference TRACK
REF : in NATURAL ); -- Reference track number

-- Builds an ABSOLUTE POLYGON REGION whose vertices are absolute
-- GLOBAL_POSITIONs
procedure BUILD_ABSOLUTE_POLYGON_REGION
( TRK : in out TRACK;
AVA : in ABSOLUTE_VERTEX_ARRAY );

-- Builds a POLYGON REGION whose vertices are relative to a reference
TRACK
procedure BUILD_RELATIVE_POLYGON_REGION
```

```
( TRK : in out TRACK;
RVA : in RELATIVE_VERTEX_ARRAY;
REF : in NATURAL ); -- reference track number

-- Returns an array of the TRACK's history points as reflected in the
-- TRACK_DATABASE
procedure TRACK_HISTORY
( TRK : in TRACK;
HISTORY_PTS_ARRAY : in out GLOB_OBS_ARRAY );

-- Changes the TRACK's course and adds a new observation
-- Usually only invoked on OWNSHIP's TRACK
procedure CHANGE_COURSE
( TRK : in out TRACK;
CRS : in ANGLE );

-- Changes the TRACK's speed and adds a new observation
-- Usually only invoked on OWNSHIP's TRACK
procedure CHANGE_SPEED
( TRK : in out TRACK;
SPD : in SPEED );

-- Changes TRACK's position without recomputing course and speed
-- Used as a correction measure
procedure CHANGE_GLOBAL_POSITION
( TRK : in out TRACK;
GP : in GLOBAL_POSITION );

-- Returns TRACK number as generated by the system
function TRACK_ID_NUMBER
( TRK : TRACK ) return NATURAL;

-- Returns TRACK's IDENTITY_TYPE
function TRACK_IDENTITY
( TRK : TRACK ) return IDENTITY_TYPE;

-- Returns TRACK's AMPLIFYING_INFO
function AMPL_INFO
( TRK : TRACK ) return AMP_STR.VSTRING;

-- Returns TRACK's CLASS
```

```
function PLATFORM_CLASS
( TRK : TRACK ) return V_AND_C_STR.VSTRING;

-- Returns TRACK vessel's name
function VESSEL_NAME
( TRK : TRACK ) return V_AND_C_STR.VSTRING;

-- Returns TRACK's TRACK_CATEGORY
function TRK_CATEGORY
( TRK : TRACK ) return TRACK_CATEGORY;

-- Returns TRACK's CONTROL_TYPE
function CONTROL
( TRK : TRACK ) return CONTROL_TYPE;

-- Returns TRACK's true course as reported/calculated in its
-- MOST_RECENT_OBSERVATION
function TRUE_COURSE
( TRK : TRACK ) return ANGLE;

-- Returns TRACK's true speed as reported/calculated in its
-- MOST_RECENT_OBSERVATION
function TRUE_SPEED
( TRK : TRACK ) return SPEED;

-- Returns TRACK's true course and speed as reported/calculated in its
-- MOST_RECENT_OBSERVATION
function TRUE_VELOCITY
( TRK : TRACK ) return VELOCITY;

-- Returns target TRACK's relative motion ( course and speed ) as seen
-- from the reference TRACK
function TARGET_RELATIVE_VELOCITY
( REFERENCE_TRACK,
TARGET_TRACK : TRACK ) return VELOCITY;

-- Returns target TRACK's relative course as seen from the reference
TRACK
function RELATIVE_COURSE
( REFERENCE_TRACK,
TARGET_TRACK : TRACK ) return ANGLE;
```

```
-- Returns target TRACK's relative speed as seen from the reference
TRACK
function RELATIVE_SPEED
( REFERENCE_TRACK,
TARGET_TRACK : TRACK ) return SPEED;

-- Returns TRACK's altitude in yards
function ALTITUDE
( TRK : TRACK ) return DISTANCE;

-- Returns TRACK's current DR ( Dead Reckoning ) position as calculated
-- from its MOST_RECENT_OBSERVATION ( last known position, course,
speed,
-- and time
function CURRENT_POSITION
( TRK : TRACK ) return GLOBAL_POSITION;

-- Returns bearing to target TRACK from reference TRACK with respect to
-- reference TRACK's heading ( not true north )
function RELATIVE_BEARING
( REFERENCE_TRACK,
TARGET_TRACK : TRACK ) return ANGLE;

-- Returns bearing to target TRACK from reference TRACK with respect to
true
-- north
function TRUE_BEARING
( REFERENCE_TRACK,
TARGET_TRACK : TRACK ) return ANGLE;

-- Returns TRACK's last entered GLOBAL_OBSERVATION
function MOST_RECENT_OBSERVATION
( TRK : TRACK ) return GLOBAL_OBSERVATION;

-- Returns category of SPECIAL_POINT TRACK
function SPEC_POINT_CATEGORY
( TRK : TRACK ) return SPECIAL_POINT_CATEGORY;

-- Returns a GLOBAL_OBSERVATION based on TRACK's relative position to
-- reference TRACK. The TRACK's course and speed are calculated based
-- on its new position and its MOST_RECENT_OBSERVATION
```

```
function MAKE_GLOBAL_OBSERVATION
( OWNSHIP_TRACK : TRACK;
TARGET_TRACK : TRACK;
TGT_REL_POS : RELATIVE_POSITION ) return GLOBAL_OBSERVATION;

-- Returns category of REGION TRACK
function REGION_CATEG
( TRK : TRACK ) return REGION_CATEGORY;

-- Returns method of REGION placement ( ABSOLUTE, RELATIVE_TO_TRACK )
function REGION_PLCMT
( TRK : TRACK ) return REGION_PLACEMENT;

-- Returns radius of CIRCLE REGION in yards
function CIRCLE_RADIUS
( TRK : TRACK ) return DISTANCE;

-- Returns location of ABSOLUTE CIRCLE REGION center
function ABS_CIRCLE_CENTER
( TRK : TRACK ) return GLOBAL_POSITION;

-- Returns bearing and range from reference TRACK to RELATIVE CIRCLE
REGION
-- center
function REL_CIRCLE_CENTER
( TRK : TRACK ) return RELATIVE_POSITION;

-- Returns all waypoints of a PATH
function PATH_POINTS
( TRK : TRACK ) return WAYPOINT_ARRAY;

-- Return location of and time to waypoint
function WAYPNT
( TRK : TRACK ) return WAYPOINT_TYPE;

-- Returns all vertices ( bearings and ranges from reference TRACK ) of
a
-- RELATIVE POLYGON REGION
function REL_REGION_VERTICES
( TRK : TRACK ) return RELATIVE_VERTEX_ARRAY;
```

```
-- Returns all vertices ( earth coordinates ) of an ABSOLUTE POLYGON
REGION
function ABS_REGION_VERTICES
( TRK : TRACK ) return ABSOLUTE_VERTEX_ARRAY;

-- Returns reference TRACK number of a RELATIVE CIRCLE REGION
function RELATIVE_CIRCLE_REFERENCE_TRK_NUM
( TRK : TRACK ) return NATURAL;

-- Returns the position of the reference TRACK of a RELATIVE CIRCLE
REGION
function RELATIVE_CIRCLE_REFERENCE_TRK_POS
( TRK : TRACK ) return GLOBAL_POSITION;

-- Returns reference TRACK number of a RELATIVE POLYGON REGION
function RELATIVE_REGION_REFERENCE_TRK_NUM
( TRK : TRACK ) return NATURAL;

-- Returns the position of the reference TRACK of a RELATIVE POLYGON
REGION
function RELATIVE_REGION_REFERENCE_TRK_POS
( TRK : TRACK ) return GLOBAL_POSITION;

-- Updates position of RELATIVE CIRCLE REGION's reference TRACK
procedure UPDATE_RELATIVE_CIRCLE_REFERENCE_TRK_POS
( TRK : in out TRACK;
GP : in GLOBAL_POSITION );

-- Updates position of RELATIVE POLYGON REGION's reference TRACK
procedure UPDATE_RELATIVE_REGION_REFERENCE_TRK_POS
( TRK : in out TRACK;
GP : in GLOBAL_POSITION );

pragma INLINE ( CREATE_TRACK, DELETE_TRACK_AND_SEND_TO_HISTORY,
     CREATE_TRACK_FILES, WRITE_TRACK_ARCHIVES_TO_TEXT_FILE,
     ADD_TRACK_OBSERVATION, SET_TRACK_IDENTITY, SET_AMPL_INFO,
     SET_PLATFORM_CLASS, SET_VESSEL_NAME, SET_ALTITUDE,
     SET_CONTROL, CHANGE_TRACK_CATEGORY,
     BUILD_WAYPOINT_SPECIAL_POINT, BUILD_NAV_HAZARD_SPECIAL_POINT,
     BUILD_GENERAL_SPECIAL_POINT, BUILD_PATH,
     BUILD_ABSOLUTE_CIRCLE_REGION, BUILD_RELATIVE_CIRCLE_REGION,
     BUILD_ABSOLUTE_POLYGON_REGION, BUILD_RELATIVE_POLYGON_REGION,
```

```
     TRACK_HISTORY, CHANGE_COURSE, CHANGE_SPEED,
     CHANGE_GLOBAL_POSITION, TRACK_ID_NUMBER, TRACK_IDENTITY,
     AMPL_INFO, PLATFORM_CLASS, VESSEL_NAME, TRK_CATEGORY, CONTROL,
     TRUE_COURSE, TRUE_SPEED, TRUE_VELOCITY,
     TARGET_RELATIVE_VELOCITY, RELATIVE_COURSE, RELATIVE_SPEED,
     ALTITUDE, CURRENT_POSITION, RELATIVE_BEARING, TRUE_BEARING,
     MOST_RECENT_OBSERVATION, SPEC_POINT_CATEGORY,
     MAKE_GLOBAL_OBSERVATION, REGION_CATEG, REGION_PLCMT,
     CIRCLE_RADIUS, ABS_CIRCLE_CENTER, REL_CIRCLE_CENTER,
     PATH_POINTS, WAYPNT, REL_REGION_VERTICES, ABS_REGION_VERTICES,
     RELATIVE_CIRCLE_REFERENCE_TRK_NUM,
     RELATIVE_CIRCLE_REFERENCE_TRK_POS,
     RELATIVE_REGION_REFERENCE_TRK_NUM,
     RELATIVE_REGION_REFERENCE_TRK_POS,
     UPDATE_RELATIVE_CIRCLE_REFERENCE_TRK_POS,
     UPDATE_RELATIVE_REGION_REFERENCE_TRK_POS );

private

 type SPECIAL_POINT_TYPE
 ( S_P_CAT : SPECIAL_POINT_CATEGORY := GENERAL ) is
 record
 case S_P_CAT is
 when WAYPOINT =>
 WAYPT : WAYPOINT_TYPE;
 when others =>
 null;
 end case;
 end record;

 type PATH_TYPE
 ( PTS : NUM_PATH_PTS := 0 ) is
 record
 WAYPTS : WAYPOINT_ARRAY ( 0 .. PTS );
 end record;

 type REL_VERTEX_TYPE
 ( PTS : NUM_VERTICES := 0 ) is
 record
 VERTICES : RELATIVE_VERTEX_ARRAY ( 0 .. PTS );
 end record;
```

```
type ABS_VERTEX_TYPE
( PTS : NUM_VERTICES := 0 ) is
record
VERTICES : ABSOLUTE_VERTEX_ARRAY ( 0 .. PTS );
end record;

type REGION_TYPE
( REG_CAT : REGION_CATEGORY := CIRCLE;
REG_PLACEMT : REGION_PLACEMENT := ABSOLUTE ) is
record
case REG_CAT is
when CIRCLE =>
RADIUS : DISTANCE;-- Circle radius
case REG_PLACEMT is
when ABSOLUTE =>
ABS_CENTER : GLOBAL_POSITION;-- Circle center posit
when RELATIVE_TO_TRACK =>
REL_CENTER : RELATIVE_POSITION;-- Circle center posit
             -- relative to ref trk
REFERENCE_TRACK1 : NATURAL; -- Ref track number
REF_TRK_POSITION1 : GLOBAL_POSITION;-- Ref track position
end case;
when POLYGON =>
case REG_PLACEMT is
when ABSOLUTE =>
ABS_VERTICES : ABS_VERTEX_TYPE;-- Vertex positions
when RELATIVE_TO_TRACK =>
REL_VERTICES : REL_VERTEX_TYPE;-- Vertex positions
             -- relative to ref trk
REFERENCE_TRACK2 : NATURAL; -- Ref track number
REF_TRK_POSITION2 : GLOBAL_POSITION;-- Ref track position
end case;
end case;
end record;

type TRACK_TYPE
( CATEGORY : TRACK_CATEGORY := UNKNOWN ) is
record
TRACK_ID : NATURAL;-- Track number
AMPL_INFO : AMP_STR.VSTRING := AMP_STR.NUL;
```

```
CONTROL : CONTROL_TYPE := LOCAL;

case CATEGORY is
when SURFACE_PLATFORM | SUBSURFACE_PLATFORM =>
S_CLASS : V_AND_C_STR.VSTRING;-- Vessel class name
S_ID : IDENTITY_TYPE := UNKNOWN;
V_NAME : V_AND_C_STR.VSTRING;-- Vessel's name
when AIR_PLATFORM =>
A_CLASS : V_AND_C_STR.VSTRING;-- Aircraft class name
A_ID : IDENTITY_TYPE := UNKNOWN;
ALTITUDE : DISTANCE;
when SPECIAL_POINT =>
S_P_TYPE : SPECIAL_POINT_TYPE;
when PATH =>
P_TYPE : PATH_TYPE;
when REGION =>
R_TYPE : REGION_TYPE;
when others =>
null;
end case;
end record;

subtype SURFACE_TRACK_TYPE is TRACK_TYPE ( SURFACE_PLATFORM );
subtype SUBSURFACE_TRACK_TYPE is TRACK_TYPE (SUBSURFACE_PLATFORM);
subtype AIR_TRACK_TYPE is TRACK_TYPE ( AIR_PLATFORM );
subtype REGION_TRACK_TYPE is TRACK_TYPE ( REGION );
subtype PATH_TRACK_TYPE is TRACK_TYPE ( PATH );
subtype SPECIAL_POINT_TRACK_TYPE is TRACK_TYPE ( SPECIAL_POINT );
subtype MAN_IN_WATER_TRACK_TYPE is TRACK_TYPE ( MAN_IN_WATER );
subtype NON_DISPLAYABLE_TRACK_TYPE is TRACK_TYPE ( NON_DISPLAYABLE );

-- Linked list structure that stores a TRACK's GLOBAL_OBSERVATIONs
type TRACK_OBS;
type TRACK_OBS_PTR is access TRACK_OBS;
type TRACK_OBS is
record
GLO_OBS : GLOBAL_OBSERVATION;
NEXT_OBS : TRACK_OBS_PTR;
end record;

type TRACK is
```

```
 record
 TRACK_DATA : TRACK_TYPE;
 TRK_OBS : TRACK_OBS_PTR;-- Pointer to first
               -- observation
 end record;

 -- Structure used to write TRACK observations to DIRECT_IO file
 type T_OBS is
 record
 T_NUM : NATURAL; -- Track number
 G_O : GLOBAL_OBSERVATION;
 end record;

 package TRACK_DATA_OUT is new DIRECT_IO ( TRACK_TYPE );
 package TRACK_OBS_OUT is new DIRECT_IO ( T_OBS );
 use TRACK_DATA_OUT, TRACK_OBS_OUT;

end TRACK_PKG;


--
========================================================================
--
-- Authors : Richard T. Irwin
--  Willie K. Bolick
--
-- Date : 29 August 1991
--
--
========================================================================

with UNCHECKED_DEALLOCATION, RELATIVE_TIME_PKG, VECTOR_2_PKG, DIRECT_IO,
 MATH, TEXT_IO;

use RELATIVE_TIME_PKG, VECTOR_2_PKG, TEXT_IO;

package body TRACK_PKG is

--
................................CREATE_TRACK............................
 procedure CREATE_TRACK
 ( GO : in GLOBAL_OBSERVATION;
```

```
LAST_TRACK_ID : in out NATURAL; -- Track number ( global var )
TRK : out TRACK ) is

T_O : TRACK_OBS_PTR;
NEW_TRK : TRACK;

begin

NEW_TRK.TRACK_DATA.TRACK_ID := LAST_TRACK_ID;
T_O := new TRACK_OBS;
T_O.GLO_OBS := GO;
NEW_TRK.TRK_OBS := T_O;
LAST_TRACK_ID := LAST_TRACK_ID + 1;-- Increment for next TRACK
TRK := NEW_TRK;

end CREATE_TRACK;

--
.......................DELETE_TRACK_AND_SEND_TO_HISTORY................
procedure DELETE_TRACK_AND_SEND_TO_HISTORY
( TRK : in out TRACK ) is

procedure FREE_OBS is
new UNCHECKED_DEALLOCATION ( TRACK_OBS, TRACK_OBS_PTR );

T1, T2 : TRACK_OBS_PTR;
T_DATA : TRACK_TYPE;
T_O : T_OBS;

TRK_FILE : TRACK_DATA_OUT.FILE_TYPE;-- File for TRACK_DATA
OBS_FILE : TRACK_OBS_OUT.FILE_TYPE;-- File for TRACK observations

T_INDEX : NATURAL;-- Index counter for TRK_FILE
O_INDEX : NATURAL;-- Index counter for OBS_FILE

begin

-- Open DIRECT_IO archive files
TRACK_DATA_OUT.OPEN ( TRK_FILE, INOUT_FILE, "TRK_FILE" );
TRACK_OBS_OUT.OPEN ( OBS_FILE, INOUT_FILE, "OBS_FILE" );
```

```
-- Get sizes of both files & set write indices to their sizes + 1
T_INDEX := NATURAL ( TRACK_DATA_OUT.SIZE ( TRK_FILE ) ) + 1;
O_INDEX := NATURAL ( TRACK_OBS_OUT.SIZE ( OBS_FILE ) ) + 1;

T_DATA := TRK.TRACK_DATA;

-- Write TRACK_DATA to file
TRACK_DATA_OUT.WRITE ( TRK_FILE, T_DATA, TRACK_DATA_OUT.POSITIVE_COUNT
( T_INDEX ) );

-- Get pointer to first TRACK observation
T1 := TRK.TRK_OBS;

-- Assign TRACK number to TRACK observation node about to be written so
-- it can later be retrieved & correlated to its TRACK_DATA
T_O.T_NUM := T_DATA.TRACK_ID;

-- Write all TRACK observations to file, freeing allocated memory along
-- the way
while T1 /= null loop
T_O.G_O := T1.GLO_OBS;
TRACK_OBS_OUT.WRITE ( OBS_FILE, T_O, TRACK_OBS_OUT.POSITIVE_COUNT
( O_INDEX ) );
O_INDEX := O_INDEX + 1;
T2 := T1.NEXT_OBS;
FREE_OBS ( T1 );
T1 := T2;
end loop;

TRACK_DATA_OUT.CLOSE ( TRK_FILE );
TRACK_OBS_OUT.CLOSE ( OBS_FILE );

end DELETE_TRACK_AND_SEND_TO_HISTORY;

--
..............................CREATE_TRACK_FILES.......................
procedure CREATE_TRACK_FILES is

TRK_FILE : TRACK_DATA_OUT.FILE_TYPE;
OBS_FILE : TRACK_OBS_OUT.FILE_TYPE;
```

```
begin

TRACK_DATA_OUT.CREATE ( TRK_FILE, INOUT_FILE, "TRK_FILE" );
TRACK_OBS_OUT.CREATE ( OBS_FILE, INOUT_FILE, "OBS_FILE" );
TRACK_DATA_OUT.CLOSE ( TRK_FILE );
TRACK_OBS_OUT.CLOSE ( OBS_FILE );

end CREATE_TRACK_FILES;

--
.......................WRITE_TRACK_ARCHIVES_TO_TEXT_FILE...............
procedure WRITE_TRACK_ARCHIVES_TO_TEXT_FILE is

T_DATA : TRACK_TYPE;
T_O : T_OBS;
TRK_NUM : NATURAL;-- Track number
FINISHED : BOOLEAN := FALSE;-- Flag to show when finished writing
T_CAT : TRACK_CATEGORY;
AMP_INFO : AMP_STR.VSTRING;
CTL : CONTROL_TYPE;
CLASS : V_AND_C_STR.VSTRING;
NAME : V_AND_C_STR.VSTRING;
IDENT : IDENTITY_TYPE;
SPEC_PT : SPECIAL_POINT_CATEGORY;
GLO_POS : GLOBAL_POSITION;
REL_POS : RELATIVE_POSITION;
ABS_TIME : ABSOLUTE_TIME;
NAT_NUM : NATURAL;
LAT_DIR : NORTH_SOUTH;
LONG_DIR : EAST_WEST;
LAT_D,
LAT_M,
LAT_S : NATURAL;
LONG_D,
LONG_M,
LONG_S : NATURAL;
Y, M, D : NATURAL;
S  : FLOAT;
REG_CAT : REGION_CATEGORY;
REG_PL : REGION_PLACEMENT;
DASHES : STRING ( 1 .. 80 ) := ( OTHERS => '=' );
```

```
DOTS : STRING ( 1 .. 80 ) := ( OTHERS => '.' );

TRK_FILE : TRACK_DATA_OUT.FILE_TYPE;
OBS_FILE : TRACK_OBS_OUT.FILE_TYPE;
TEXT_FILE : TEXT_IO.FILE_TYPE;

--
+++++++++++++++++++++++++++++++PRINT_GLOBAL_POSITION+++++++++++++++++++++
-- Prints TRACK observation points as earth coordinates to text file
procedure PRINT_GLOBAL_POSITION is

begin

GET_LATITUDE ( GLO_POS, LAT_DIR, LAT_D, LAT_M, LAT_S );
GET_LONGITUDE ( GLO_POS, LONG_DIR, LONG_D, LONG_M, LONG_S );

PUT ( TEXT_FILE, NATURAL'IMAGE ( LAT_D ) );
PUT ( TEXT_FILE, NATURAL'IMAGE ( LAT_M ) );
PUT ( TEXT_FILE, NATURAL'IMAGE ( LAT_S ) );

if LAT_DIR = N then
PUT ( TEXT_FILE, " N " );
else
PUT ( TEXT_FILE, " S " );
end if;

PUT ( TEXT_FILE, NATURAL'IMAGE ( LONG_D ) );
PUT ( TEXT_FILE, NATURAL'IMAGE ( LONG_M ) );
PUT ( TEXT_FILE, NATURAL'IMAGE ( LONG_S ) );

if LONG_DIR = W then
PUT ( TEXT_FILE, " W" );
else
PUT ( TEXT_FILE, " E" );
end if;

end PRINT_GLOBAL_POSITION;

--
++++++++++++++++++++++++++++++PRINT_OBSERVATION_TIME++++++++++++++++++++
-- Prints time of TRACK observation as mm/dd/yy hh:mm:ss
```

```
procedure PRINT_OBSERVATION_TIME is

begin

Y := YEAR ( ABS_TIME );
M := MONTH ( ABS_TIME );
D := DAY ( ABS_TIME );
S := TIME_OF_DAY ( ABS_TIME );

PUT ( TEXT_FILE, NATURAL'IMAGE ( M ) );
PUT ( TEXT_FILE, "/" );
PUT ( TEXT_FILE, NATURAL'IMAGE ( D ) );
PUT ( TEXT_FILE, "/" );
PUT ( TEXT_FILE, NATURAL'IMAGE ( Y - 1900 ) );
PUT ( TEXT_FILE, " " );
PUT ( TEXT_FILE, NATURAL'IMAGE ( HOURS ( TIME_OF_DAY ( ABS_TIME ) ) ) );
PUT ( TEXT_FILE, ':' );
PUT ( TEXT_FILE, NATURAL'IMAGE ( MINUTES ( TIME_OF_DAY ( ABS_TIME ) ) )
);
PUT ( TEXT_FILE, ':' );
PUT ( TEXT_FILE, NATURAL'IMAGE ( NATURAL ( SECONDS ( TIME_OF_DAY
( ABS_TIME ) ) ) ) );

end PRINT_OBSERVATION_TIME;

--
==========================================================================
begin -- WRITE_TRACK_ARCHIVES_TO_TEXT_FILE

-- Open DIRECT_IO TRACK archive files
TRACK_DATA_OUT.OPEN ( TRK_FILE, INOUT_FILE, "TRK_FILE" );
TRACK_OBS_OUT.OPEN ( OBS_FILE, INOUT_FILE, "OBS_FILE" );

-- Create text file for TRACK history
TEXT_IO.CREATE ( TEXT_FILE, NAME => "TRACKS.HIS" );

while NOT TRACK_DATA_OUT.END_OF_FILE ( TRK_FILE ) loop
-- Read in all unique TRACK_DATA records one at a time

TRACK_DATA_OUT.READ ( TRK_FILE, T_DATA );
TRK_NUM := T_DATA.TRACK_ID;-- Get TRACK number to identify its
```

```
        -- observations in OBS_FILE

-- Read in & write TRACK_DATA info to text file

T_CAT := T_DATA.CATEGORY;
AMP_INFO := T_DATA.AMPL_INFO;
PUT_LINE ( TEXT_FILE, DASHES );
PUT ( TEXT_FILE, "TRACK NUMBER :" );
PUT ( TEXT_FILE, NATURAL'IMAGE ( TRK_NUM ) );
SET_COL ( TEXT_FILE, 40 );
PUT ( TEXT_FILE, "CONTROL : " );

if T_DATA.CONTROL = LINK then
PUT ( TEXT_FILE, "LINK" );
else
PUT ( TEXT_FILE, "LOCAL" );
end if;

TEXT_IO.NEW_LINE ( TEXT_FILE );
PUT ( TEXT_FILE, "AMPLIFYING INFO : " );
PUT ( TEXT_FILE, AMP_STR.STR ( AMP_INFO ) );
TEXT_IO.NEW_LINE ( TEXT_FILE );
PUT ( TEXT_FILE, "TRACK CATEGORY : " );

case T_CAT is

when UNKNOWN =>
PUT ( TEXT_FILE, "UNKNOWN" );
TEXT_IO.NEW_LINE ( TEXT_FILE );
PUT_LINE ( TEXT_FILE, DASHES );

when SURFACE_PLATFORM =>
CLASS := T_DATA.S_CLASS;
NAME := T_DATA.V_NAME;
IDENT := T_DATA.S_ID;
PUT ( TEXT_FILE, "SURFACE_PLATFORM" );
TEXT_IO.NEW_LINE ( TEXT_FILE );
PUT ( TEXT_FILE, "CLASS : " );
PUT ( TEXT_FILE, V_AND_C_STR.STR ( CLASS ) );
TEXT_IO.NEW_LINE ( TEXT_FILE );
PUT ( TEXT_FILE, "IDENTITY : " );
```

```
case IDENT is
when UNKNOWN =>
  PUT ( TEXT_FILE, "UNKNOWN" );
when FRIENDLY =>
PUT ( TEXT_FILE, "FRIENDLY" );
when HOSTILE =>
  PUT ( TEXT_FILE, "HOSTILE" );
when NEUTRAL =>
  PUT ( TEXT_FILE, "NEUTRAL" );
end case;

  TEXT_IO.NEW_LINE ( TEXT_FILE );
  PUT ( TEXT_FILE, "VESSEL NAME : " );
  PUT ( TEXT_FILE, V_AND_C_STR.STR ( NAME ) );
  TEXT_IO.NEW_LINE ( TEXT_FILE );
  PUT_LINE ( TEXT_FILE, DASHES );

when SUBSURFACE_PLATFORM =>
  PUT ( TEXT_FILE, "SUBSURFACE_PLATFORM" );
  CLASS := T_DATA.S_CLASS;
  NAME := T_DATA.V_NAME;
  IDENT := T_DATA.S_ID;
  TEXT_IO.NEW_LINE ( TEXT_FILE );
  PUT ( TEXT_FILE, "CLASS : " );
  PUT ( TEXT_FILE, V_AND_C_STR.STR ( CLASS ) );
  TEXT_IO.NEW_LINE ( TEXT_FILE );
  PUT ( TEXT_FILE, "IDENTITY : " );

  case IDENT is
  when UNKNOWN =>
  PUT ( TEXT_FILE, "UNKNOWN" );
  when FRIENDLY =>
  PUT ( TEXT_FILE, "FRIENDLY" );
  when HOSTILE =>
  PUT ( TEXT_FILE, "HOSTILE" );
  when NEUTRAL =>
  PUT ( TEXT_FILE, "NEUTRAL" );
  end case;

  TEXT_IO.NEW_LINE ( TEXT_FILE );
```

```
      PUT ( TEXT_FILE, "VESSEL NAME : " );
      PUT ( TEXT_FILE, V_AND_C_STR.STR ( NAME ) );
      TEXT_IO.NEW_LINE ( TEXT_FILE );
   PUT_LINE ( TEXT_FILE, DASHES );

   when AIR_PLATFORM =>
      PUT ( TEXT_FILE, "AIR_PLATFORM" );
      CLASS := T_DATA.A_CLASS;
      IDENT := T_DATA.A_ID;
      TEXT_IO.NEW_LINE ( TEXT_FILE );
      PUT ( TEXT_FILE, "CLASS : " );
      PUT ( TEXT_FILE, V_AND_C_STR.STR ( CLASS ) );
      TEXT_IO.NEW_LINE ( TEXT_FILE );
      PUT ( TEXT_FILE, "IDENTITY : " );

      case IDENT is
      when UNKNOWN =>
      PUT ( TEXT_FILE, "UNKNOWN" );
      when FRIENDLY =>
      PUT ( TEXT_FILE, "FRIENDLY" );
      when HOSTILE =>
      PUT ( TEXT_FILE, "HOSTILE" );
      when NEUTRAL =>
      PUT ( TEXT_FILE, "NEUTRAL" );
      end case;

      TEXT_IO.NEW_LINE ( TEXT_FILE );
   PUT_LINE ( TEXT_FILE, DASHES );

   when REGION =>
      REG_CAT := T_DATA.R_TYPE.REG_CAT;
      REG_PL := T_DATA.R_TYPE.REG_PLACEMT;
      PUT ( TEXT_FILE, "REGION" );
      SET_COL ( TEXT_FILE, 35 );

      case reg_cat is

      when CIRCLE =>
      PUT ( TEXT_FILE, "CIRCLE" );
      SET_COL ( TEXT_FILE, 45 );
```

```
case REG_PL is

when ABSOLUTE =>
GLO_POS := T_DATA.R_TYPE.ABS_CENTER;
  PUT ( TEXT_FILE, "ABSOLUTE" );
  TEXT_IO.NEW_LINE ( TEXT_FILE );
  PUT ( TEXT_FILE,"CIRCLE CENTER :" );
  PRINT_GLOBAL_POSITION;

when RELATIVE_TO_TRACK =>
  PUT ( TEXT_FILE, "RELATIVE TO TRACK" );
  NAT_NUM := T_DATA.R_TYPE.REFERENCE_TRACK1;
  PUT ( TEXT_FILE,NATURAL'IMAGE ( NAT_NUM ) );
  TEXT_IO.NEW_LINE ( TEXT_FILE );
  PUT_LINE ( TEXT_FILE, "BRG / RG FROM" );
  PUT ( TEXT_FILE, "REFERENCE TRACK :");
  NAT_NUM := NATURAL ( RADIANS_TO_DEGREES ( BEARING_TO
     ( T_DATA.R_TYPE.REL_CENTER ) ) );
  PUT ( TEXT_FILE,NATURAL'IMAGE(NAT_NUM));
  PUT ( TEXT_FILE, '/' );
  NAT_NUM := NATURAL ( RANGE_OF ( T_DATA.R_TYPE.REL_CENTER ) );
  PUT ( TEXT_FILE,NATURAL'IMAGE ( NAT_NUM ) );

end case;

TEXT_IO.NEW_LINE ( TEXT_FILE );
PUT ( TEXT_FILE, "CIRCLE RADIUS :" );
NAT_NUM := NATURAL ( T_DATA.R_TYPE.RADIUS );
PUT ( TEXT_FILE, NATURAL'IMAGE ( NAT_NUM ) );
TEXT_IO.NEW_LINE ( TEXT_FILE );

when POLYGON =>
PUT ( TEXT_FILE, "POLYGON" );
SET_COL ( TEXT_FILE, 45 );

case REG_PL is

when ABSOLUTE =>
  PUT ( TEXT_FILE, "ABSOLUTE" );
  TEXT_IO.NEW_LINE ( TEXT_FILE );
  PUT ( TEXT_FILE, "POLYGON VERTICES :" );
```

```
    TEXT_IO.NEW_LINE ( TEXT_FILE );
  NAT_NUM := NATURAL ( T_DATA.R_TYPE.ABS_VERTICES.PTS );

    for I in 0 .. NAT_NUM loop
    GLO_POS := T_DATA.R_TYPE.ABS_VERTICES.VERTICES (I);
    PRINT_GLOBAL_POSITION;
    TEXT_IO.NEW_LINE ( TEXT_FILE );
    end loop;

  when RELATIVE_TO_TRACK =>
    PUT ( TEXT_FILE, "RELATIVE TO TRACK" );
    NAT_NUM := T_DATA.R_TYPE.REFERENCE_TRACK1;
    PUT ( TEXT_FILE,NATURAL'IMAGE ( NAT_NUM ) );
    TEXT_IO.NEW_LINE ( TEXT_FILE );
    PUT_LINE(TEXT_FILE,"POLYGON VERTICES" );
    PUT ( TEXT_FILE, "(BRG/RG FM REF TRK) :");
    TEXT_IO.NEW_LINE ( TEXT_FILE );
    NAT_NUM := NATURAL ( T_DATA.R_TYPE.REL_VERTICES.PTS );

    for I in 0 .. NAT_NUM loop
    REL_POS := T_DATA.R_TYPE.REL_VERTICES.VERTICES (I);
    NAT_NUM := NATURAL ( RADIANS_TO_DEGREES ( BEARING_TO
        ( REL_POS ) ) );
    PUT ( TEXT_FILE, NATURAL'IMAGE ( NAT_NUM ) );
    PUT ( TEXT_FILE, '/' );
    NAT_NUM := NATURAL ( RANGE_OF ( REL_POS ) );
    PUT ( TEXT_FILE, NATURAL'IMAGE ( NAT_NUM ) );
    TEXT_IO.NEW_LINE ( TEXT_FILE );
    end loop;

  end case;

  end case;

PUT_LINE ( TEXT_FILE, DASHES );

when PATH =>
  PUT ( TEXT_FILE, "PATH" );
  TEXT_IO.NEW_LINE ( TEXT_FILE );
  NAT_NUM := T_DATA.P_TYPE.PTS;
```

```
   for I in 0 .. NAT_NUM loop
   GLO_POS := T_DATA.P_TYPE.WAYPTS ( I ).POSITION;
   ABS_TIME := T_DATA.P_TYPE.WAYPTS ( I ).TIME_TO;
   PUT ( TEXT_FILE, "PATH POINT POSITION :" );
   PRINT_GLOBAL_POSITION;
   TEXT_IO.NEW_LINE ( TEXT_FILE );
PUT ( TEXT_FILE, "TIME TO PATH POINT :" );
   PRINT_OBSERVATION_TIME;
   TEXT_IO.NEW_LINE ( TEXT_FILE );
   end loop;

PUT_LINE ( TEXT_FILE, DASHES );

when SPECIAL_POINT =>
   SPEC_PT := T_DATA.S_P_TYPE.S_P_CAT;
   PUT ( TEXT_FILE, "SPECIAL_POINT" );
SET_COL ( TEXT_FILE, 40 );

   case SPEC_PT is

   when GENERAL =>
   PUT ( TEXT_FILE, "GENERAL" );

   when WAYPOINT =>
   GLO_POS := T_DATA.S_P_TYPE.WAYPT.POSITION;
   ABS_TIME := T_DATA.S_P_TYPE.WAYPT.TIME_TO;
   PUT ( TEXT_FILE, "WAYPOINT" );
   TEXT_IO.NEW_LINE ( TEXT_FILE );
   PUT ( TEXT_FILE, "WAYPOINT POSITION :" );
   PRINT_GLOBAL_POSITION;
   TEXT_IO.NEW_LINE ( TEXT_FILE );
PUT ( TEXT_FILE, "TIME TO WAYPOINT :" );
   PRINT_OBSERVATION_TIME;

   when NAV_HAZARD =>
   PUT ( TEXT_FILE, "NAV_HAZARD" );

   end case;

   TEXT_IO.NEW_LINE ( TEXT_FILE );
PUT_LINE ( TEXT_FILE, DASHES );
```

```
when MAN_IN_WATER =>
   PUT ( TEXT_FILE, "MAN_IN_WATER" );
   TEXT_IO.NEW_LINE ( TEXT_FILE );
PUT_LINE ( TEXT_FILE, DASHES );

when NON_DISPLAYABLE =>
   PUT ( TEXT_FILE, "NON_DISPLAYABLE" );
   TEXT_IO.NEW_LINE ( TEXT_FILE );
PUT_LINE ( TEXT_FILE, DASHES );

end case;

-- Since we know the TRACK number of the current TRACK being read/
written,
-- we can now identify its observations in OBS_FILE by searching on its
-- TRACK number. Also, since a TRACK and its observations are dropped
-- at the same time, the observations for any particular TRACK will be
-- contiguous in the file.
while NOT TRACK_OBS_OUT.END_OF_FILE ( OBS_FILE ) loop

exit when FINISHED;

TRACK_OBS_OUT.READ ( OBS_FILE, T_O );

  if TRK_NUM = T_O.T_NUM then
  -- A match on TRACK number is found in the OBS_FILE,

   -- All observations will be together, so keep reading until a
mismatch
   -- is found
   while NOT FINISHED loop

   -- Read in & write all TRACK's observations

   GLO_POS := T_O.G_O.POSITION;
   ABS_TIME := T_O.G_O.OBSERVATION_TIME;
   PUT ( TEXT_FILE, "OBSERVATION POSITION :" );
   PRINT_GLOBAL_POSITION;
   TEXT_IO.NEW_LINE ( TEXT_FILE );
   PUT ( TEXT_FILE, "TIME OF OBSERVATION :" );
```

```
   PRINT_OBSERVATION_TIME;
   TEXT_IO.NEW_LINE ( TEXT_FILE );
   PUT ( TEXT_FILE, "OBSERVED COURSE :" );
   NAT_NUM := NATURAL ( RADIANS_TO_DEGREES ( COURSE
( T_O.G_O.COURSE_AND_SPEED ) ) );
   PUT ( TEXT_FILE, NATURAL'IMAGE ( NAT_NUM ) );
   TEXT_IO.NEW_LINE ( TEXT_FILE );
   PUT ( TEXT_FILE, "OBSERVED SPEED :" );
   NAT_NUM := NATURAL ( SPEED_IN_KNOTS ( SPD
( T_O.G_O.COURSE_AND_SPEED ) ) );
   PUT ( TEXT_FILE, NATURAL'IMAGE ( NAT_NUM ) );
   TEXT_IO.NEW_LINE ( TEXT_FILE );
   PUT_LINE ( TEXT_FILE, DOTS );

if NOT TRACK_OBS_OUT.END_OF_FILE (OBS_FILE) then

   -- Get next TRACK observation
   TRACK_OBS_OUT.READ ( OBS_FILE, T_O );

   if TRK_NUM /= T_O.T_NUM then

     -- Mismatch on TRACK number
   FINISHED := TRUE;

     -- Write next TRACK info on new page
   TEXT_IO.NEW_PAGE ( TEXT_FILE );

   end if;

   else -- No more TRACK observations

   FINISHED := TRUE;
   TEXT_IO.NEW_PAGE ( TEXT_FILE );

end if;

end loop;

end if;

end loop;
```

```
-- Go back to the start of OBS_FILE to start reading observations for
-- the next TRACK
TRACK_OBS_OUT.RESET ( OBS_FILE );

-- Reset flag
FINISHED := FALSE;

end loop;

TRACK_DATA_OUT.CLOSE ( TRK_FILE );
TRACK_OBS_OUT.CLOSE ( OBS_FILE );
TEXT_IO.CLOSE ( TEXT_FILE );

end WRITE_TRACK_ARCHIVES_TO_TEXT_FILE;

--
..............................ADD_TRACK_OBSERVATION....................
procedure ADD_TRACK_OBSERVATION
( TRK : in out TRACK;
GO : in GLOBAL_OBSERVATION ) is

T_O : TRACK_OBS_PTR;

begin

-- Add observation to head of list
T_O := new TRACK_OBS;
T_O.GLO_OBS := GO;
T_O.NEXT_OBS := TRK.TRK_OBS;
TRK.TRK_OBS := T_O;

end ADD_TRACK_OBSERVATION;

--
..............................SET_TRACK_IDENTITY.......................
procedure SET_TRACK_IDENTITY
( TRK : in out TRACK;
TID : in IDENTITY_TYPE ) is

begin
```

```
case TRK.TRACK_DATA.CATEGORY is
when SURFACE_PLATFORM | SUBSURFACE_PLATFORM =>
 TRK.TRACK_DATA.S_ID := TID;
when AIR_PLATFORM =>
 TRK.TRACK_DATA.A_ID := TID;
when others =>
 null;
end case;

end SET_TRACK_IDENTITY;

--
................................SET_AMPL_INFO.........................
procedure SET_AMPL_INFO
( TRK : out TRACK;
AMP : in AMP_STR.VSTRING ) is
begin
TRK.TRACK_DATA.AMPL_INFO := AMP;
end SET_AMPL_INFO;

--
.............................SET_PLATFORM_CLASS......................
procedure SET_PLATFORM_CLASS
( TRK : in out TRACK;
PC : in V_AND_C_STR.VSTRING ) is

begin

case TRK.TRACK_DATA.CATEGORY is
when SURFACE_PLATFORM | SUBSURFACE_PLATFORM =>
 TRK.TRACK_DATA.S_CLASS := PC;
when AIR_PLATFORM =>
 TRK.TRACK_DATA.A_CLASS := PC;
when others =>
 null;
end case;

end SET_PLATFORM_CLASS;

--
..............................SET_VESSEL_NAME.......................
```

```
procedure SET_VESSEL_NAME
( TRK : in out TRACK;
VES : in V_AND_C_STR.VSTRING ) is

begin

if ( TRK.TRACK_DATA.CATEGORY = SURFACE_PLATFORM ) or
( TRK.TRACK_DATA.CATEGORY = SUBSURFACE_PLATFORM ) then
TRK.TRACK_DATA.V_NAME := VES;
end if;

end SET_VESSEL_NAME;

--
.................................SET_ALTITUDE...........................
procedure SET_ALTITUDE
( TRK : in out TRACK;
ALT : in DISTANCE ) is

begin

if TRK.TRACK_DATA.CATEGORY = AIR_PLATFORM then
TRK.TRACK_DATA.ALTITUDE := ALT;
end if;

end SET_ALTITUDE;

--
.................................SET_CONTROL............................
procedure SET_CONTROL
( TRK : out TRACK;
CON : in CONTROL_TYPE ) is

begin
TRK.TRACK_DATA.CONTROL := CON;
end SET_CONTROL;

--
..............................CHANGE_TRACK_CATEGORY....................
procedure CHANGE_TRACK_CATEGORY
( TRK1 : in out TRACK;
CAT : in TRACK_CATEGORY ) is
```

```
TRK2 : TRACK;
SFC : SURFACE_TRACK_TYPE;
SUB : SUBSURFACE_TRACK_TYPE;
AIR : AIR_TRACK_TYPE;
REG : REGION_TRACK_TYPE;
SPP : SPECIAL_POINT_TRACK_TYPE;
PTH : PATH_TRACK_TYPE;
MIW : MAN_IN_WATER_TRACK_TYPE;
NON : NON_DISPLAYABLE_TRACK_TYPE;

begin

case CAT is
when SURFACE_PLATFORM =>
 TRK2.TRACK_DATA := SFC;
when SUBSURFACE_PLATFORM =>
 TRK2.TRACK_DATA := SUB;
when AIR_PLATFORM =>
 TRK2.TRACK_DATA := AIR;
when REGION =>
 TRK2.TRACK_DATA := REG;
when SPECIAL_POINT =>
 TRK2.TRACK_DATA := SPP;
when PATH =>
 TRK2.TRACK_DATA := PTH;
when MAN_IN_WATER =>
 TRK2.TRACK_DATA := MIW;
when NON_DISPLAYABLE =>
 TRK2.TRACK_DATA := NON;
when others =>
 null;
end case;

TRK2.TRACK_DATA.TRACK_ID := TRK1.TRACK_DATA.TRACK_ID;
TRK2.TRACK_DATA.AMPL_INFO := TRK1.TRACK_DATA.AMPL_INFO;
TRK2.TRACK_DATA.CONTROL := TRK1.TRACK_DATA.CONTROL;
TRK2.TRK_OBS := TRK1.TRK_OBS;
TRK1 := TRK2;

end CHANGE_TRACK_CATEGORY;
```

```
--
..........................BUILD_WAYPOINT_SPECIAL_POINT................
procedure BUILD_WAYPOINT_SPECIAL_POINT
( TRK : in out TRACK;
POS : in GLOBAL_POSITION;
TYME : in ABSOLUTE_TIME ) is

WP : SPECIAL_POINT_TYPE ( WAYPOINT );

begin

CHANGE_TRACK_CATEGORY ( TRK, SPECIAL_POINT );
WP.WAYPT.POSITION := POS;
WP.WAYPT.TIME_TO := TYME;
TRK.TRACK_DATA.S_P_TYPE := WP;

end BUILD_WAYPOINT_SPECIAL_POINT;

--
...........................BUILD_NAV_HAZARD_SPECIAL_POINT.............
procedure BUILD_NAV_HAZARD_SPECIAL_POINT
( TRK : in out TRACK ) is

NH : SPECIAL_POINT_TYPE ( NAV_HAZARD );

begin

CHANGE_TRACK_CATEGORY ( TRK, SPECIAL_POINT );
TRK.TRACK_DATA.S_P_TYPE := NH;

end BUILD_NAV_HAZARD_SPECIAL_POINT;

--
..........................BUILD_GENERAL_SPECIAL_POINT.................
procedure BUILD_GENERAL_SPECIAL_POINT
( TRK : in out TRACK ) is

GEN : SPECIAL_POINT_TYPE;

begin
```

```
CHANGE_TRACK_CATEGORY ( TRK, SPECIAL_POINT );
TRK.TRACK_DATA.S_P_TYPE := GEN;

end BUILD_GENERAL_SPECIAL_POINT;

--
..................................BUILD_PATH..........................
procedure BUILD_PATH
( TRK : in out TRACK;
PTS : in WAYPOINT_ARRAY ) is

N : NUM_PATH_PTS := PTS'LAST;
PTH : PATH_TYPE ( N );

begin

CHANGE_TRACK_CATEGORY ( TRK, PATH );
PTH.WAYPTS := PTS;
TRK.TRACK_DATA.P_TYPE := PTH;

end BUILD_PATH;

--
..........................BUILD_ABSOLUTE_CIRCLE_REGION................
procedure BUILD_ABSOLUTE_CIRCLE_REGION
( TRK : in out TRACK;
RAD : in DISTANCE;
CTR : in GLOBAL_POSITION ) is

ABS_CIRCLE : REGION_TYPE;

begin

CHANGE_TRACK_CATEGORY ( TRK, REGION );
ABS_CIRCLE.RADIUS := RAD;
ABS_CIRCLE.ABS_CENTER := CTR;
TRK.TRACK_DATA.R_TYPE := ABS_CIRCLE;

end BUILD_ABSOLUTE_CIRCLE_REGION;
```

```
--
.........................BUILD_RELATIVE_CIRCLE_REGION.................
procedure BUILD_RELATIVE_CIRCLE_REGION
( TRK : in out TRACK;
RAD : in DISTANCE;
CTR : in RELATIVE_POSITION;
REF : in NATURAL ) is

REL_CIRCLE : REGION_TYPE ( CIRCLE, RELATIVE_TO_TRACK );

begin

CHANGE_TRACK_CATEGORY ( TRK, REGION );
REL_CIRCLE.RADIUS := RAD;
REL_CIRCLE.REL_CENTER := CTR;
REL_CIRCLE.REFERENCE_TRACK1 := REF;
TRK.TRACK_DATA.R_TYPE := REL_CIRCLE;

end BUILD_RELATIVE_CIRCLE_REGION;

--
.........................BUILD_ABSOLUTE_POLYGON_REGION................
procedure BUILD_ABSOLUTE_POLYGON_REGION
( TRK : in out TRACK;
AVA : in ABSOLUTE_VERTEX_ARRAY ) is

N : NUM_VERTICES := AVA'LAST;
AV_TYPE : ABS_VERTEX_TYPE ( N );
ABS_POLY : REGION_TYPE ( POLYGON, ABSOLUTE );

begin

CHANGE_TRACK_CATEGORY ( TRK, REGION );
AV_TYPE.VERTICES := AVA;
ABS_POLY.ABS_VERTICES := AV_TYPE;
TRK.TRACK_DATA.R_TYPE := ABS_POLY;

end BUILD_ABSOLUTE_POLYGON_REGION;

--
........................BUILD_RELATIVE_POLYGON_REGION.................
procedure BUILD_RELATIVE_POLYGON_REGION
```

```
( TRK : in out TRACK;
RVA : in RELATIVE_VERTEX_ARRAY;
REF : in NATURAL ) is

N : NUM_VERTICES := RVA'LAST;
RV_TYPE : REL_VERTEX_TYPE ( N );
REL_POLY : REGION_TYPE ( POLYGON, RELATIVE_TO_TRACK );

begin

CHANGE_TRACK_CATEGORY ( TRK, REGION );
RV_TYPE.VERTICES := RVA;
REL_POLY.REL_VERTICES := RV_TYPE;
REL_POLY.REFERENCE_TRACK2 := REF;
TRK.TRACK_DATA.R_TYPE := REL_POLY;

end BUILD_RELATIVE_POLYGON_REGION;

--
..............................TRACK_HISTORY............................
procedure TRACK_HISTORY
( TRK : in TRACK;
HISTORY_PTS_ARRAY : in out GLOB_OBS_ARRAY ) is

-- Points to first TRACK observation
NEXT_OBSERVATION_PTR : TRACK_OBS_PTR := TRK.TRK_OBS;

begin

-- Read in as many observations as the user requested ( as indicated by
-- the size of the array
for I in HISTORY_PTS_ARRAY'RANGE loop

-- If there are less TRACK observations than the user requested
if NEXT_OBSERVATION_PTR = null then
return;
end if;

-- Fill array element with current observation
HISTORY_PTS_ARRAY ( I ) := NEXT_OBSERVATION_PTR.GLO_OBS;
```

```
-- Point to next observation
NEXT_OBSERVATION_PTR := NEXT_OBSERVATION_PTR.NEXT_OBS;

end loop;

end TRACK_HISTORY;

--
...............................CHANGE_COURSE..........................
procedure CHANGE_COURSE
( TRK : in out TRACK;
CRS : in ANGLE ) is

-- TRACK's current speed
TRUE_SPD : SPEED := TRUE_SPEED ( TRK );

-- TRACK's current position
TRK_POS : GLOBAL_POSITION := CURRENT_POSITION ( TRK );

NEW_OBS : GLOBAL_OBSERVATION;
NEW_CRS_SPD : VELOCITY;

begin

NEW_CRS_SPD := MAKE_VELOCITY ( TRUE_SPD, CRS );

NEW_OBS.OBSERVATION_TIME := NOW;
NEW_OBS.POSITION := TRK_POS;
NEW_OBS.COURSE_AND_SPEED := NEW_CRS_SPD;

-- Since we're changing TRACK's course, need to add a new observation
ADD_TRACK_OBSERVATION ( TRK, NEW_OBS );

end CHANGE_COURSE;

--
.................................CHANGE_SPEED.........................
procedure CHANGE_SPEED
( TRK : in out TRACK;
SPD : in SPEED ) is
```

```
-- TRACK's current course
TRUE_CRS : ANGLE := TRUE_COURSE ( TRK );

-- TRACK's current position
TRK_POS : GLOBAL_POSITION := CURRENT_POSITION ( TRK );

NEW_OBS : GLOBAL_OBSERVATION;
NEW_CRS_SPD : VELOCITY;

begin

NEW_CRS_SPD := MAKE_VELOCITY ( SPD, TRUE_CRS );

NEW_OBS.OBSERVATION_TIME := NOW;
NEW_OBS.POSITION := TRK_POS;
NEW_OBS.COURSE_AND_SPEED := NEW_CRS_SPD;

-- Since we're changing TRACK's speed, need to add a new observation
ADD_TRACK_OBSERVATION ( TRK, NEW_OBS );

end CHANGE_SPEED;

--
.............................CHANGE_GLOBAL_POSITION....................
procedure CHANGE_GLOBAL_POSITION
( TRK : in out TRACK;
GP : in GLOBAL_POSITION ) is

-- TRACK's current course and speed
TRUE_VEL : VELOCITY := TRUE_VELOCITY ( TRK );

NEW_OBS : GLOBAL_OBSERVATION;

begin

NEW_OBS.OBSERVATION_TIME := NOW;
NEW_OBS.COURSE_AND_SPEED := TRUE_VEL;
NEW_OBS.POSITION := GP;

-- Since we're changing TRACK's course and speed, need to add a new
-- observation
```

```
ADD_TRACK_OBSERVATION ( TRK, NEW_OBS );

end CHANGE_GLOBAL_POSITION;

--
..............................TRACK_ID_NUMBER.........................
function TRACK_ID_NUMBER
( TRK : TRACK ) return NATURAL is

begin
return TRK.TRACK_DATA.TRACK_ID;
end TRACK_ID_NUMBER;

--
...............................TRACK_IDENTITY.........................
function TRACK_IDENTITY
( TRK : TRACK ) return IDENTITY_TYPE is

begin

case TRK.TRACK_DATA.CATEGORY is
when SURFACE_PLATFORM | SUBSURFACE_PLATFORM =>
return TRK.TRACK_DATA.S_ID;
when AIR_PLATFORM =>
return TRK.TRACK_DATA.A_ID;
when others =>
null;
end case;

end TRACK_IDENTITY;

--
.................................AMPL_INFO............................
function AMPL_INFO
( TRK : TRACK ) return AMP_STR.VSTRING is

begin
return TRK.TRACK_DATA.AMPL_INFO;
end AMPL_INFO;
```

```
--
................................PLATFORM_CLASS........................
function PLATFORM_CLASS
( TRK : TRACK ) return V_AND_C_STR.VSTRING is

begin

case TRK.TRACK_DATA.CATEGORY is
when SURFACE_PLATFORM | SUBSURFACE_PLATFORM =>
return TRK.TRACK_DATA.S_CLASS;
when AIR_PLATFORM =>
return TRK.TRACK_DATA.A_CLASS;
when others =>
null;
end case;

end PLATFORM_CLASS;

--
................................VESSEL_NAME...........................
function VESSEL_NAME
( TRK : TRACK ) return V_AND_C_STR.VSTRING is

begin

if ( TRK.TRACK_DATA.CATEGORY = SURFACE_PLATFORM ) or
( TRK.TRACK_DATA.CATEGORY = SUBSURFACE_PLATFORM ) then
return TRK.TRACK_DATA.V_NAME;
end if;

end VESSEL_NAME;

--
................................TRK_CATEGORY..........................
function TRK_CATEGORY
( TRK : TRACK ) return TRACK_CATEGORY is

begin
return TRK.TRACK_DATA.CATEGORY;
end TRK_CATEGORY;
```

```
--
..................................CONTROL...................................
function CONTROL
( TRK : TRACK ) return CONTROL_TYPE is

begin
return TRK.TRACK_DATA.CONTROL;
end CONTROL;

--
................................TRUE_COURSE.................................
function TRUE_COURSE
( TRK : TRACK ) return ANGLE is

begin
return COURSE ( MOST_RECENT_OBSERVATION ( TRK ).COURSE_AND_SPEED );
end TRUE_COURSE;

--
.................................TRUE_SPEED.................................
function TRUE_SPEED
( TRK : TRACK ) return SPEED is

begin
return SPD ( MOST_RECENT_OBSERVATION ( TRK ).COURSE_AND_SPEED );
end TRUE_SPEED;

--
...............................TRUE_VELOCITY...............................
function TRUE_VELOCITY
( TRK : TRACK ) return VELOCITY is

begin
return MOST_RECENT_OBSERVATION ( TRK ).COURSE_AND_SPEED;
end TRUE_VELOCITY;

--
............................TARGET_RELATIVE_VELOCITY..................
function TARGET_RELATIVE_VELOCITY
( REFERENCE_TRACK,
TARGET_TRACK : TRACK ) return VELOCITY is
```

```
REF_TRUE_VELOCITY,
TGT_TRUE_VELOCITY : VELOCITY;

begin

-- Get target & reference TRACK's true velocity
REF_TRUE_VELOCITY := TRUE_VELOCITY ( REFERENCE_TRACK );
TGT_TRUE_VELOCITY := TRUE_VELOCITY ( TARGET_TRACK );

-- The difference in the 2 true velocity vectors gives relative velocity
return VECTOR_2_PKG."-" ( TGT_TRUE_VELOCITY, REF_TRUE_VELOCITY );

end TARGET_RELATIVE_VELOCITY;

--
...............................RELATIVE_COURSE.........................
function RELATIVE_COURSE
( REFERENCE_TRACK,
TARGET_TRACK : TRACK ) return ANGLE is

begin
return COURSE ( TARGET_RELATIVE_VELOCITY
( REFERENCE_TRACK, TARGET_TRACK ) );
end RELATIVE_COURSE;

--
...............................RELATIVE_SPEED..........................
function RELATIVE_SPEED
( REFERENCE_TRACK,                                                     .
TARGET_TRACK : TRACK ) return SPEED is

begin
return SPD ( TARGET_RELATIVE_VELOCITY ( REFERENCE_TRACK, TARGET_TRACK )
);
end RELATIVE_SPEED;

--
.................................ALTITUDE..............................
function ALTITUDE
( TRK : TRACK ) return DISTANCE is

begin
```

```
if TRK.TRACK_DATA.CATEGORY = AIR_PLATFORM then
return TRK.TRACK_DATA.ALTITUDE;
end if;

end ALTITUDE;

--
..............................CURRENT_POSITION........................
function CURRENT_POSITION
( TRK : TRACK ) return GLOBAL_POSITION is

TIME_DIFFERENCE : RELATIVE_TIME;
TRACK_SPEED : SPEED := TRUE_SPEED ( TRK );
TRACK_COURSE : ANGLE := TRUE_COURSE ( TRK );
DEAD_RECKONING_DISTANCE : DISTANCE;
DEAD_RECKONING_POSITION : RELATIVE_POSITION;
LAST_GLOBAL_POSITION : GLOBAL_POSITION;

begin

-- Get time difference between last TRACK observation and now in order
to
-- compute distance traveled
TIME_DIFFERENCE := NOW - MOST_RECENT_OBSERVATION ( TRK
).OBSERVATION_TIME;

-- Compute distance traveled based on last known speed and time
difference
DEAD_RECKONING_DISTANCE := TRACK_SPEED * TIME_DIFFERENCE;

-- Make a RELATIVE_POSITION vector
DEAD_RECKONING_POSITION := RELATIVE_POSITION ( MAKE_POLAR_VECTOR_2 (
FLOAT
( DEAD_RECKONING_DISTANCE ), TRACK_COURSE ) );

-- Get TRACK's last known GLOBAL_POSITION
LAST_GLOBAL_POSITION := MOST_RECENT_OBSERVATION ( TRK ).POSITION;

-- We can now find the TRACK's current position based on last
-- GLOBAL_POSITION and the relative position from that point
return FIND_GLOBAL_POSITION ( DEAD_RECKONING_POSITION,
```

```
LAST_GLOBAL_POSITION );

end CURRENT_POSITION;

--
..............................RELATIVE_BEARING........................
function RELATIVE_BEARING
( REFERENCE_TRACK,
TARGET_TRACK : TRACK ) return ANGLE is

REFERENCE_TRUE_COURSE : ANGLE := TRUE_COURSE ( REFERENCE_TRACK );
REFERENCE_POSITION : GLOBAL_POSITION := CURRENT_POSITION
( REFERENCE_TRACK );
TARGET_POSITION : GLOBAL_POSITION := CURRENT_POSITION
( TARGET_TRACK );
BEARING_TO_TARGET : ANGLE;
REL_BEARING : ANGLE;

begin

-- Relative bearing to a target means we assume reference TRACK's
-- heading to be 000.0 ( no matter what course it is actually on ).
-- The target TRACK's relative bearing from the reference TRACK is a
-- function of the target TRACK's true bearing from the reference TRACK
-- and the reference TRACK's true course.

-- Get true bearing to the target
BEARING_TO_TARGET := BEARING_TO ( FIND_RELATIVE_POSITION
          ( TARGET_POSITION, REFERENCE_POSITION ) );

-- Compute relative bearing
REL_BEARING := MATH.PI * 2.0 - REFERENCE_TRUE_COURSE +
BEARING_TO_TARGET;

-- Correct for angle > 360.0
if REL_BEARING >= MATH.PI * 2.0 then
REL_BEARING := REL_BEARING - MATH.PI * 2.0;
end if;

return REL_BEARING;
```

```
end RELATIVE_BEARING;

--
...............................TRUE_BEARING............................
function TRUE_BEARING
( REFERENCE_TRACK,
TARGET_TRACK : TRACK ) return ANGLE is

REFERENCE_POSITION : GLOBAL_POSITION := CURRENT_POSITION
( REFERENCE_TRACK );
TARGET_POSITION : GLOBAL_POSITION := CURRENT_POSITION
( TARGET_TRACK );

begin

return BEARING_TO ( FIND_RELATIVE_POSITION
     ( TARGET_POSITION, REFERENCE_POSITION ) );

end TRUE_BEARING;

--
..........................MOST_RECENT_OBSERVATION.....................
function MOST_RECENT_OBSERVATION
( TRK : TRACK ) return GLOBAL_OBSERVATION is

begin
return TRK.TRK_OBS.GLO_OBS;
end MOST_RECENT_OBSERVATION;

--
.............................SPEC_POINT_CATEGORY......................
function SPEC_POINT_CATEGORY
( TRK : TRACK ) return SPECIAL_POINT_CATEGORY is

begin
return TRK.TRACK_DATA.S_P_TYPE.S_P_CAT;
end SPEC_POINT_CATEGORY;

--
............................MAKE_GLOBAL_OBSERVATION...................
function MAKE_GLOBAL_OBSERVATION
( OWNSHIP_TRACK : TRACK;
```

```
TARGET_TRACK : TRACK;
TGT_REL_POS : RELATIVE_POSITION ) return GLOBAL_OBSERVATION is

GO : GLOBAL_OBSERVATION;
OP : GLOBAL_POSITION := CURRENT_POSITION ( OWNSHIP_TRACK );
GP_1,
GP_2 : GLOBAL_POSITION;
TP : TRACK_OBS_PTR := TARGET_TRACK.TRK_OBS;
CRS_1 : ANGLE;
SPD_1 : SPEED;
RP_1 : RELATIVE_POSITION;
RT : RELATIVE_TIME;

begin

-- Get target TRACK's position based on reference TRACK's position
-- and the target's relative position from the reference
GP_1 := FIND_GLOBAL_POSITION ( TGT_REL_POS, OP );

GO.POSITION := GP_1;
GO.OBSERVATION_TIME := NOW;

-- In order to compute course and speed, we need at least 1 previous
-- observation with which to compare against its new observation

if TP = null then -- No previous observations
GO.COURSE_AND_SPEED := MAKE_VELOCITY ( 0.0, 0.0 );
else
GP_2 := TP.GLO_OBS.POSITION;

-- Compute time difference between last observation and new one
RT := GO.OBSERVATION_TIME - TP.GLO_OBS.OBSERVATION_TIME;

-- Find the position difference between the 2 observations
RP_1 := FIND_RELATIVE_POSITION ( GP_1, GP_2 );

-- Get the new course and speed
CRS_1 := BEARING_TO ( RP_1 );
SPD_1 := RANGE_OF ( RP_1 ) / RT;

GO.COURSE_AND_SPEED := MAKE_VELOCITY ( SPD_1, CRS_1 );
```

```
end if;

return GO;

end MAKE_GLOBAL_OBSERVATION;

--
.................................REGION_CATEG........................
function REGION_CATEG
( TRK : TRACK ) return REGION_CATEGORY is

begin

if TRK_CATEGORY ( TRK ) = REGION then
return TRK.TRACK_DATA.R_TYPE.REG_CAT;
end if;

end REGION_CATEG;

--
.................................REGION_PLCMT........................
function REGION_PLCMT
( TRK : TRACK ) return REGION_PLACEMENT is

begin

if TRK_CATEGORY ( TRK ) = REGION then
return TRK.TRACK_DATA.R_TYPE.REG_PLACEMT;
end if;

end REGION_PLCMT;

--
.................................CIRCLE_RADIUS.......................
function CIRCLE_RADIUS
( TRK : TRACK ) return DISTANCE is

begin

if ( TRK_CATEGORY ( TRK ) = REGION ) and then
( TRK.TRACK_DATA.R_TYPE.REG_CAT = CIRCLE ) then
```

```
return TRK.TRACK_DATA.R_TYPE.RADIUS;
end if;

end CIRCLE_RADIUS;

--
..............................ABS_CIRCLE_CENTER........................
function ABS_CIRCLE_CENTER
( TRK : TRACK ) return GLOBAL_POSITION is

begin

if ( TRK_CATEGORY ( TRK ) = REGION ) and then
( REGION_CATEG ( TRK ) = CIRCLE ) and then
( REGION_PLCMT ( TRK ) = ABSOLUTE ) then
return TRK.TRACK_DATA.R_TYPE.ABS_CENTER;
end if;

end ABS_CIRCLE_CENTER;

--
..............................REL_CIRCLE_CENTER........................
function REL_CIRCLE_CENTER
( TRK : TRACK ) return RELATIVE_POSITION is

begin

if ( TRK_CATEGORY ( TRK ) = REGION ) and then
( REGION_CATEG ( TRK ) = CIRCLE ) and then
( REGION_PLCMT ( TRK ) = RELATIVE_TO_TRACK ) then
return TRK.TRACK_DATA.R_TYPE.REL_CENTER;
end if;

end REL_CIRCLE_CENTER;

--
.................................PATH_POINTS...........................
function PATH_POINTS
( TRK : TRACK ) return WAYPOINT_ARRAY is

begin
```

```
if TRK_CATEGORY ( TRK ) = PATH then
return TRK.TRACK_DATA.P_TYPE.WAYPTS;
end if;
end PATH_POINTS;

--
.................................WAYPNT...............................
function WAYPNT
( TRK : TRACK ) return WAYPOINT_TYPE is

begin

if ( TRK_CATEGORY ( TRK ) = SPECIAL_POINT ) and then
( SPEC_POINT_CATEGORY ( TRK ) = WAYPOINT ) then
return TRK.TRACK_DATA.S_P_TYPE.WAYPT;
end if;

end WAYPNT;

--
............................REL_REGION_VERTICES......................
function REL_REGION_VERTICES
( TRK : TRACK ) return RELATIVE_VERTEX_ARRAY is

begin

if ( TRK_CATEGORY ( TRK ) = REGION ) and then
( REGION_CATEG ( TRK ) = POLYGON ) and then
( REGION_PLCMT ( TRK ) = RELATIVE_TO_TRACK ) then
return TRK.TRACK_DATA.R_TYPE.REL_VERTICES.VERTICES;
end if;

end REL_REGION_VERTICES;

--
............................ABS_REGION_VERTICES......................
function ABS_REGION_VERTICES
( TRK : TRACK ) return ABSOLUTE_VERTEX_ARRAY is

begin

if ( TRK_CATEGORY ( TRK ) = REGION ) and then
```

```
( REGION_CATEG ( TRK ) = POLYGON ) and then
( REGION_PLCMT ( TRK ) = ABSOLUTE ) then
return TRK.TRACK_DATA.R_TYPE.ABS_VERTICES.VERTICES;
end if;

end ABS_REGION_VERTICES;

--
........................RELATIVE_CIRCLE_REFERENCE_TRK_NUM...............
function RELATIVE_CIRCLE_REFERENCE_TRK_NUM
( TRK : TRACK ) return NATURAL is

begin

if ( TRK_CATEGORY ( TRK ) = REGION ) and then
( REGION_CATEG ( TRK ) = CIRCLE ) and then
( REGION_PLCMT ( TRK ) = RELATIVE_TO_TRACK ) then
return TRK.TRACK_DATA.R_TYPE.REFERENCE_TRACK1;
end if;

end RELATIVE_CIRCLE_REFERENCE_TRK_NUM;

--
.......................RELATIVE_CIRCLE_REFERENCE_TRK_POS...............
function RELATIVE_CIRCLE_REFERENCE_TRK_POS
( TRK : TRACK ) return GLOBAL_POSITION is

begin

if ( TRK_CATEGORY ( TRK ) = REGION ) and then
( REGION_CATEG ( TRK ) = CIRCLE ) and then
( REGION_PLCMT ( TRK ) = RELATIVE_TO_TRACK ) then
return TRK.TRACK_DATA.R_TYPE.REF_TRK_POSITION1;
end if;

end RELATIVE_CIRCLE_REFERENCE_TRK_POS;

--
........................RELATIVE_REGION_REFERENCE_TRK_NUM...............
function RELATIVE_REGION_REFERENCE_TRK_NUM
( TRK : TRACK ) return NATURAL is

begin
```

```
  if ( TRK_CATEGORY ( TRK ) = REGION ) and then
 ( REGION_CATEG ( TRK ) = POLYGON ) and then
 ( REGION_PLCMT ( TRK ) = RELATIVE_TO_TRACK ) then
 return TRK.TRACK_DATA.R_TYPE.REFERENCE_TRACK2;
 end if;

 end RELATIVE_REGION_REFERENCE_TRK_NUM;

......................RELATIVE_REGION_REFERENCE_TRK_POS.................
 function RELATIVE_REGION_REFERENCE_TRK_POS
 ( TRK : TRACK ) return GLOBAL_POSITION is

 begin

 if ( TRK_CATEGORY ( TRK ) = REGION ) and then
 ( REGION_CATEG ( TRK ) = POLYGON ) and then
 ( REGION_PLCMT ( TRK ) = RELATIVE_TO_TRACK ) then
 return TRK.TRACK_DATA.R_TYPE.REF_TRK_POSITION2;
 end if;

 end RELATIVE_REGION_REFERENCE_TRK_POS;

...................UPDATE_RELATIVE_CIRCLE_REFERENCE_TRK_POS.............
 procedure UPDATE_RELATIVE_CIRCLE_REFERENCE_TRK_POS
 ( TRK : in out TRACK;
 GP : in GLOBAL_POSITION ) is

 begin
 TRK.TRACK_DATA.R_TYPE.REF_TRK_POSITION1 := GP;
 end UPDATE_RELATIVE_CIRCLE_REFERENCE_TRK_POS;

--
...................UPDATE_RELATIVE_REGION_REFERENCE_TRK_POS.............
 procedure UPDATE_RELATIVE_REGION_REFERENCE_TRK_POS
 ( TRK : in out TRACK;
 GP : in GLOBAL_POSITION ) is

 begin
 TRK.TRACK_DATA.R_TYPE.REF_TRK_POSITION2 := GP;
 end UPDATE_RELATIVE_REGION_REFERENCE_TRK_POS;
........................................................................
end TRACK_PKG;
```

APPENDIX D

FILTER PACKAGE

```
--
==========================================================================
--
-- Authors : Richard T. Irwin
--  Willie K. Bolick
--
-- Date : 29 August 1991
--
--
...........................................................................
--
-- Description : Defines abstract data type FILTER and associated
--  functions & procedures
--
--
==========================================================================

with TRACK_PKG, DISTANCE_PKG, ABSOLUTE_TIME_PKG, DIRECT_IO;

use TRACK_PKG, DISTANCE_PKG, ABSOLUTE_TIME_PKG;

package FILTER_PKG is

 -- An ATOMIC_FILTER is based on 1 of the 3 below attributes
 type FILTER_CATEGORY is
 ( DISTANCE_FILTER,
 TRACK_CATEGORY_FILTER,
 PLATFORM_IDENTITY_FILTER );

 -- An ATOMIC_FILTER based on DISTANCE_FILTER is further based on the
 -- attributes below
 type DISTANCE_ATTRIBUTE_ID is
 ( RANGE_FROM_REFERENCE_TRACK,
 ALTITUDE ); -- from ownship

 type RELATION_ID is
```

```
( EQUAL, NOT_EQUAL, LESS, LESS_OR_EQUAL, GREATER, GREATER_OR_EQUAL );

subtype EQUALITY_RELATION_ID is
RELATION_ID range EQUAL .. NOT_EQUAL;

-- Each AND_FILTER is a set of ATOMIC_FILTERs
type ATOMIC_FILTER
( FILTER_TYPE : FILTER_CATEGORY := DISTANCE_FILTER ) is private;

-- a track passes an AND_FILTER iff it passes every ATOMIC_FILTER in
-- the list.
type AND_FILTER is private;

-- a track passes a FILTER iff it passes at least one AND_FILTER in
-- the list.
type FILTER is private;

-- Makes an ATOMIC_FILTER based on DISTANCE attributes
procedure MAKE_DISTANCE_ATOMIC_FILTER
( DAF_ATTRIB_ID : in DISTANCE_ATTRIBUTE_ID;
DAF_LIMIT : in DISTANCE;
DAF_REF_TRACK : in TRACK;
DAF_RELATION : in RELATION_ID;
ATOMIC_FILTUR : out ATOMIC_FILTER );

-- Makes an ATOMIC_FILTER based on TRACK_CATEGORY attributes
procedure MAKE_TRACK_CATEGORY_ATOMIC_FILTER
( TCAF_DESIRED_TRK_CAT : in TRACK_CATEGORY;
TCAF_EQ_REL_ID : in EQUALITY_RELATION_ID;
ATOMIC_FILTUR : out ATOMIC_FILTER );

-- Makes an ATOMIC_FILTER based on IDENTITY_TYPE attributes
procedure MAKE_PLATFORM_IDENTITY_ATOMIC_FILTER
( PIAF_DESIRED_PLAT_ID : in IDENTITY_TYPE;
PIAF_EQ_REL_ID : in EQUALITY_RELATION_ID;
ATOMIC_FILTUR : out ATOMIC_FILTER );

-- Once the ATOMIC_FILTER is built, it is added to the current
AND_FILTER
procedure ADD_ATOMIC_FILTER_TO_AND_FILTER
( ATOMIC_FILTUR : in ATOMIC_FILTER;
```

```
AND_FILTUR : in out AND_FILTER );

-- Once the AND_FILTER is filled with desired ATOMIC_FILTERs, it is
added to
-- the FILTER
procedure ADD_AND_FILTER_TO_FILTER
( AND_FILTUR : in out AND_FILTER;
FILTUR : in out FILTER );

-- Clears the old FILTER to make way for a new one
procedure CLEAR_FILTER
( F : in out FILTER );

-- Creates a DIRECT_IO file that stores all FILTERs used during the
session
procedure CREATE_FILTER_FILE;

-- Once a new FILTER is created, it is written to the file created in
the
-- above procedure
procedure WRITE_FILTER
( F : in FILTER );

-- Compares a TRACK to the current FILTER to determine whether or not to
-- pass it to the TACPLOT ( user display )
function TEST_FILTER
( F : FILTER;
T : TRACK ) return BOOLEAN;

-- Everything in the active database is passed to TACPLOT
function EVERYTHING return FILTER;

-- Retrieves all FILTERs written to DIRECT_IO file and writes them to a
-- human readable text file for historical purposes
procedure WRITE_FILTER_ARCHIVES_TO_TEXT_FILE;

pragma INLINE ( MAKE_DISTANCE_ATOMIC_FILTER,
     MAKE_TRACK_CATEGORY_ATOMIC_FILTER,
     MAKE_PLATFORM_IDENTITY_ATOMIC_FILTER,
     ADD_ATOMIC_FILTER_TO_AND_FILTER, ADD_AND_FILTER_TO_FILTER,
     CLEAR_FILTER, WRITE_FILTER, TEST_FILTER, EVERYTHING );
```

```
private

 type ATOMIC_FILTER
 ( FILTER_TYPE : FILTER_CATEGORY := DISTANCE_FILTER ) is
 record
 case FILTER_TYPE is
 when DISTANCE_FILTER =>
 D_ATTRIB_ID : DISTANCE_ATTRIBUTE_ID;
 D_LIMIT : DISTANCE;
 REFERENCE_TRACK : TRACK;
 D_RELATION : RELATION_ID;
 when TRACK_CATEGORY_FILTER =>
 DESIRED_TRK_CAT : TRACK_CATEGORY;
 EQ_REL_ID1 : EQUALITY_RELATION_ID;
 when PLATFORM_IDENTITY_FILTER =>
 DESIRED_PLAT_ID : IDENTITY_TYPE;
 EQ_REL_ID2 : EQUALITY_RELATION_ID;
 end case;
 end record;

 -- Data structure used to link up all ATOMIC_FILTERs of an AND_FILTER
 type ATOMIC_FILTER_NODE;
 type ATOMIC_FILTER_PTR is access ATOMIC_FILTER_NODE;
 type ATOMIC_FILTER_NODE is
 record
 ATM_FILTER : ATOMIC_FILTER;
 NEXT_ATOMIC_FILTER : ATOMIC_FILTER_PTR;
 end record;

 type AND_FILTER is
 record
 FIRST_ATOMIC_FILTER : ATOMIC_FILTER_PTR;
 end record;

 -- Data structure used to link up all AND_FILTERs of a FILTER
 type AND_FILTER_NODE;
 type AND_FILTER_PTR is access AND_FILTER_NODE;
 type AND_FILTER_NODE is
 record
 AND_FLTR : AND_FILTER;
 NEXT_AND_FILTER : AND_FILTER_PTR;
```

```
 end record;

 type FILTER is
 record
 FIRST_AND_FILTER : AND_FILTER_PTR;
 end record;

 -- Each ATOMIC_FILTER within the FILTER is written to the DIRECT_IO file
 -- in the record format below
 type ATOMIC_FILTER_OUT is
 record
 FILTER_NUM : POSITIVE;-- Number of the FILTER that the
            -- ATOMIC_FILTER belongs to
 AND_FILTER_NUM : NATURAL; -- Number of the AND_FILTER that the
            -- ATOMIC_FILTER belongs to
 ATOMIC_FILTUR : ATOMIC_FILTER;
 TIME_OUT : ABSOLUTE_TIME;-- Date & time the FILTER was written
            -- to the file
 end record;

 package FILTER_INOUT is new DIRECT_IO ( ATOMIC_FILTER_OUT );
 use FILTER_INOUT;

end FILTER_PKG;

--
========================================================================
--
-- Authors : Richard T. Irwin
--  Willie K. Bolick
--
-- Date : 29 August 1991
--
--
========================================================================

with GLOBAL_POSITION_PKG, RELATIVE_POSITION_PKG, UNCHECKED_DEALLOCATION,
 ABSOLUTE_TIME_PKG, RELATIVE_TIME_PKG, TEXT_IO;

use GLOBAL_POSITION_PKG, RELATIVE_POSITION_PKG, ABSOLUTE_TIME_PKG,
 RELATIVE_TIME_PKG;
```

```
package body FILTER_PKG is

--
.........................MAKE_DISTANCE_ATOMIC_FILTER.................
 procedure MAKE_DISTANCE_ATOMIC_FILTER
 ( DAF_ATTRIB_ID : in DISTANCE_ATTRIBUTE_ID;
 DAF_LIMIT : in DISTANCE;
 DAF_REF_TRACK : in TRACK;
 DAF_RELATION : in RELATION_ID;
 ATOMIC_FILTUR : out ATOMIC_FILTER ) is

 begin

 ATOMIC_FILTUR.D_ATTRIB_ID := DAF_ATTRIB_ID;
 ATOMIC_FILTUR.D_LIMIT := DAF_LIMIT;
 ATOMIC_FILTUR.REFERENCE_TRACK := DAF_REF_TRACK;
 ATOMIC_FILTUR.D_RELATION := DAF_RELATION;

 end MAKE_DISTANCE_ATOMIC_FILTER;

--
.......................MAKE_TRACK_CATEGORY_ATOMIC_FILTER...............
 procedure MAKE_TRACK_CATEGORY_ATOMIC_FILTER
 ( TCAF_DESIRED_TRK_CAT : in TRACK_CATEGORY;
 TCAF_EQ_REL_ID : in EQUALITY_RELATION_ID;
 ATOMIC_FILTUR : out ATOMIC_FILTER ) is

 TCAF : ATOMIC_FILTER ( TRACK_CATEGORY_FILTER );

 begin

 TCAF.DESIRED_TRK_CAT := TCAF_DESIRED_TRK_CAT;
 TCAF.EQ_REL_ID1 := TCAF_EQ_REL_ID;
 ATOMIC_FILTUR := TCAF;

 end MAKE_TRACK_CATEGORY_ATOMIC_FILTER;

--
......................MAKE_PLATFORM_IDENTITY_ATOMIC_FILTER.............
 procedure MAKE_PLATFORM_IDENTITY_ATOMIC_FILTER
 ( PIAF_DESIRED_PLAT_ID : in IDENTITY_TYPE;
```

```
PIAF_EQ_REL_ID : in EQUALITY_RELATION_ID;
ATOMIC_FILTUR : out ATOMIC_FILTER ) is

PIAF : ATOMIC_FILTER ( PLATFORM_IDENTITY_FILTER );

begin

PIAF.DESIRED_PLAT_ID := PIAF_DESIRED_PLAT_ID;
PIAF.EQ_REL_ID2 := PIAF_EQ_REL_ID;
ATOMIC_FILTUR := PIAF;

end MAKE_PLATFORM_IDENTITY_ATOMIC_FILTER;

--
..........................ADD_ATOMIC_FILTER_TO_AND_FILTER...............
procedure ADD_ATOMIC_FILTER_TO_AND_FILTER
( ATOMIC_FILTUR : in ATOMIC_FILTER;
AND_FILTUR : in out AND_FILTER ) is

ATOMIC_FILTUR_PTR : ATOMIC_FILTER_PTR;

begin

ATOMIC_FILTUR_PTR := new ATOMIC_FILTER_NODE;
ATOMIC_FILTUR_PTR.ATM_FILTER := ATOMIC_FILTUR;

-- If the newly-created ATOMIC_FILTER is the first to be added to the
-- current AND_FILTER, its position is recorded as such in the
AND_FILTER.
-- All subsequent ATOMIC_FILTERs are appended to the head of the
-- AND_FILTER linked list of ATOMIC_FILTERs

if AND_FILTUR.FIRST_ATOMIC_FILTER /= null then
ATOMIC_FILTUR_PTR.NEXT_ATOMIC_FILTER := AND_FILTUR.FIRST_ATOMIC_FILTER;
end if;

AND_FILTUR.FIRST_ATOMIC_FILTER := ATOMIC_FILTUR_PTR;

end ADD_ATOMIC_FILTER_TO_AND_FILTER;
```

```
--
.............................ADD_AND_FILTER_TO_FILTER..................
procedure ADD_AND_FILTER_TO_FILTER
( AND_FILTUR : in out AND_FILTER;
FILTUR : in out FILTER ) is

AFP : AND_FILTER_PTR;
ANF : AND_FILTER := AND_FILTUR;

begin

AFP := new AND_FILTER_NODE;
AFP.AND_FLTR := ANF;

-- If the newly-filled AND_FILTER is the first to be added to the
-- current FILTER, its position is recorded as such in the FILTER.
-- All subsequent AND_FILTERs are appended to the head of the
-- FILTER linked list of AND_FILTERs

if FILTUR.FIRST_AND_FILTER /= null then
AFP.NEXT_AND_FILTER := FILTUR.FIRST_AND_FILTER;
end if;

FILTUR.FIRST_AND_FILTER := AFP;

AND_FILTUR.FIRST_ATOMIC_FILTER := null; -- Reset for new AND_FILTER

end ADD_AND_FILTER_TO_FILTER;

--
.................................CLEAR_FILTER...........................
procedure CLEAR_FILTER
( F : in out FILTER ) is

procedure FREE_ATOMIC_FILTER is
new UNCHECKED_DEALLOCATION( ATOMIC_FILTER_NODE, ATOMIC_FILTER_PTR );

procedure FREE_AND_FILTER is
new UNCHECKED_DEALLOCATION( AND_FILTER_NODE, AND_FILTER_PTR );

ATFP  : ATOMIC_FILTER_PTR;
```

```
ANFP  : AND_FILTER_PTR;
NEXT_ATOMIC_PTR : ATOMIC_FILTER_PTR;
NEXT_AND_PTR : AND_FILTER_PTR;

begin

-- Don't bother clearing an already empty FILTER
if F.FIRST_AND_FILTER = null then
return;
else

-- Start the clear operation at the first AND_FILTER
ANFP := F.FIRST_AND_FILTER;

-- Keep clearing until no more AND_FILTERs
while ANFP /= null loop

NEXT_AND_PTR := ANFP.NEXT_AND_FILTER;

 -- Get the first ATOMIC_FILTER of this AND_FILTER
ATFP := ANFP.AND_FLTR.FIRST_ATOMIC_FILTER;

 -- Clear all the ATOMIC_FILTERs of this AND_FILTER
while ATFP /= null loop
NEXT_ATOMIC_PTR := ATFP.NEXT_ATOMIC_FILTER;
FREE_ATOMIC_FILTER ( ATFP );
ATFP := NEXT_ATOMIC_PTR;
end loop;

-- Clear the AND_FILTER
FREE_AND_FILTER ( ANFP );

 -- Get the next AND_FILTER
ANFP := NEXT_AND_PTR;

end loop;

end if;

F.FIRST_AND_FILTER := null;
```

```
end CLEAR_FILTER;

--
............................CREATE_FILTER_FILE........................
procedure CREATE_FILTER_FILE is

FILTER_FILE : FILTER_INOUT.FILE_TYPE; -- Archive file

begin

FILTER_INOUT.CREATE ( FILTER_FILE, INOUT_FILE, "FILTER_FILE" );
FILTER_INOUT.CLOSE ( FILTER_FILE );

end CREATE_FILTER_FILE;

--
...............................WRITE_FILTER..........................
procedure WRITE_FILTER
( F : in FILTER ) is

FILTER_FILE : FILTER_INOUT.FILE_TYPE; -- Archive file
FLTR_NUM : POSITIVE;-- Number of FILTERs in archive
F_INDEX : NATURAL;-- Write index
AND_FLTR_NUM : NATURAL := 1;-- Number of AND_FILTERs
ATOMIC_FLTR_OUT : ATOMIC_FILTER_OUT;-- Archive element structure
ATFP  : ATOMIC_FILTER_PTR;
ANFP  : AND_FILTER_PTR;
WRITE_TIME : ABSOLUTE_TIME := NOW;-- Time of write operation
AF_OUT : ATOMIC_FILTER_OUT;

begin

-- Open archive file & find end of file to determine where to write the
-- next FILTER
FILTER_INOUT.OPEN ( FILTER_FILE, INOUT_FILE, "FILTER_FILE" );
F_INDEX := NATURAL ( FILTER_INOUT.SIZE ( FILTER_FILE ) ) + 1;

-- Read last FILTER in file to get its FILTER number, then add 1 to
assign
-- new FILTER number
if FILTER_INOUT.SIZE ( FILTER_FILE ) > 0 then
FILTER_INOUT.READ ( FILTER_FILE, AF_OUT, POSITIVE_COUNT
```

```
     ( SIZE ( FILTER_FILE ) ) );
FLTR_NUM := AF_OUT.FILTER_NUM + 1;
else
FLTR_NUM := 1;
end if;

-- Set write index
FILTER_INOUT.SET_INDEX ( FILTER_FILE, POSITIVE_COUNT ( F_INDEX ) );

-- Get first AND_FILTER
ANFP := F.FIRST_AND_FILTER;

-- Assign values to output structure
ATOMIC_FLTR_OUT.FILTER_NUM := FLTR_NUM;
ATOMIC_FLTR_OUT.TIME_OUT := WRITE_TIME;

-- There will be no AND_FILTERs if the FILTER is set to accept all
TRACKs
if ANFP = null then
ATOMIC_FLTR_OUT.AND_FILTER_NUM := 0;
FILTER_INOUT.WRITE ( FILTER_FILE, ATOMIC_FLTR_OUT,
       POSITIVE_COUNT ( F_INDEX ) );
else

-- While there are still AND_FILTERs left to write
while ANFP /= null loop

 -- Assign AND_FILTER number to output structure
ATOMIC_FLTR_OUT.AND_FILTER_NUM := AND_FLTR_NUM;

 -- Get first ATOMIC_FILTER of this AND_FILTER
ATFP := ANFP.AND_FLTR.FIRST_ATOMIC_FILTER;

 -- While there are still ATOMIC_FILTERs left to write
while ATFP /= null loop

  -- Assign ATOMIC_FILTER to output structure
ATOMIC_FLTR_OUT.ATOMIC_FILTUR := ATFP.ATM_FILTER;

  -- Write output structure to archive file
FILTER_INOUT.WRITE ( FILTER_FILE, ATOMIC_FLTR_OUT,
```

```
       POSITIVE_COUNT ( F_INDEX ) );

   -- Increment write index
 F_INDEX := F_INDEX + 1;

   -- Get next ATOMIC_FILTER
 ATFP := ATFP.NEXT_ATOMIC_FILTER;

 end loop;

  -- Increment AND_FILTER number for next AND_FILTER
 AND_FLTR_NUM := AND_FLTR_NUM + 1;

  -- Get next AND_FILTER
 ANFP := ANFP.NEXT_AND_FILTER;

 end loop;

 end if;

 FILTER_INOUT.CLOSE ( FILTER_FILE );

 end WRITE_FILTER;

--
................................TEST_FILTER...............................
 function TEST_FILTER
 ( F : FILTER;
 T : TRACK ) return BOOLEAN is

 B : BOOLEAN := FALSE;
 AF : ATOMIC_FILTER;
 ATFP : ATOMIC_FILTER_PTR;
 ANFP : AND_FILTER_PTR;

 -- Tests input TRACK against one ATOMIC_FILTER and returns the result
 function TEST_ATOMIC_FILTER
 ( ATF : ATOMIC_FILTER ) return BOOLEAN is

 TGT_POS : GLOBAL_POSITION;
 REF_POS : GLOBAL_POSITION;
```

```
T_CATEG : TRACK_CATEGORY := TRK_CATEGORY ( T );
T_ID : IDENTITY_TYPE;

begin

case ATF.FILTER_TYPE is

  -- ATOMIC_FILTER based on distance-type attributes
when DISTANCE_FILTER =>

case ATF.D_ATTRIB_ID is

   -- Distance-type attribute is range from a reference TRACK
when RANGE_FROM_REFERENCE_TRACK =>

   -- Get reference & target positions
REF_POS := CURRENT_POSITION ( ATF.REFERENCE_TRACK );
TGT_POS := CURRENT_POSITION ( T );

case ATF.D_RELATION is

    -- Range from reference TRACK must be equal to the input
    -- parameter value in order to pass
when EQUAL =>
if RANGE_OF ( FIND_RELATIVE_POSITION
       ( TGT_POS, REF_POS ) ) = ATF.D_LIMIT then
return TRUE;
else
return FALSE;
end if;

    -- Range from reference TRACK must not be equal to the input
    -- parameter value in order to pass
when NOT_EQUAL =>
if RANGE_OF ( FIND_RELATIVE_POSITION
       ( TGT_POS, REF_POS ) ) /= ATF.D_LIMIT then
return TRUE;
else
return FALSE;
end if;
```

```
    -- Range from reference TRACK must be less than the input
    -- parameter value in order to pass
when LESS =>
if RANGE_OF ( FIND_RELATIVE_POSITION
       ( TGT_POS, REF_POS ) ) < ATF.D_LIMIT then
return TRUE;
else
return FALSE;
end if;

    -- Range from reference TRACK must be less than or equal to the
    -- input parameter value in order to pass
    when LESS_OR_EQUAL =>
     if RANGE_OF ( FIND_RELATIVE_POSITION
       ( TGT_POS, REF_POS ) ) <= ATF.D_LIMIT then
     return TRUE;
     else
     return FALSE;
     end if;

    -- Range from reference TRACK must be greater than the input
    -- parameter value in order to pass
    when GREATER =>
     if RANGE_OF ( FIND_RELATIVE_POSITION
       ( TGT_POS, REF_POS ) ) > ATF.D_LIMIT then
     return TRUE;
     else
     return FALSE;
     end if;

    -- Range from reference TRACK must be greater than or equal to
    -- the input parameter value in order to pass
    when GREATER_OR_EQUAL =>
     if RANGE_OF ( FIND_RELATIVE_POSITION
       ( TGT_POS, REF_POS ) ) >= ATF.D_LIMIT then
     return TRUE;
     else
     return FALSE;
     end if;

end case;
```

```
  -- Distance-type attribute is altitude
when ALTITUDE =>

  -- Since altitude applies only to aircraft, others will fail this
  -- test
if TRK_CATEGORY ( T ) /= AIR_PLATFORM then
return FALSE;
end if;

case ATF.D_RELATION is

   -- Altitude must be equal to the input parameter value in order
   -- to pass
when EQUAL =>
if ALTITUDE ( T ) = ATF.D_LIMIT then
return TRUE;
else
return FALSE;
end if;

   -- Altitude must not be equal to the input parameter value in
   -- order to pass
   when NOT_EQUAL =>
    if ALTITUDE ( T ) /= ATF.D_LIMIT then
    return TRUE;
    else
    return FALSE;
    end if;

   -- Altitude must be less than the input parameter value in order
   -- to pass
   when LESS =>
    if ALTITUDE ( T ) < ATF.D_LIMIT then
    return TRUE;
    else
    return FALSE;
    end if;

   -- Altitude must be less than or equal to the input parameter
   -- value in order to pass
```

```
    when LESS_OR_EQUAL =>
     if ALTITUDE ( T ) <= ATF.D_LIMIT then
     return TRUE;
     else
     return FALSE;
     end if;

    -- Altitude must be greater than the input parameter value in
    -- order to pass
    when GREATER =>
     if ALTITUDE ( T ) > ATF.D_LIMIT then
     return TRUE;
     else
     return FALSE;
     end if;

    -- Altitude must be greater than or equal to the input parameter
    -- value in order to pass
    when GREATER_OR_EQUAL =>
     if ALTITUDE ( T ) >= ATF.D_LIMIT then
     return TRUE;
     else
     return FALSE;
     end if;

end case;

  end case;

  -- ATOMIC_FILTER based on category-type attributes
when TRACK_CATEGORY_FILTER =>

case ATF.EQ_REL_ID1 is

  -- TRACK_CATEGORY must be equal to the input parameter value in
  -- order to pass
when EQUAL =>
if T_CATEG = ATF.DESIRED_TRK_CAT then
return TRUE;
else
return FALSE;
```

```
end if;

  -- TRACK_CATEGORY must not be equal to the input parameter value
  -- in order to pass
  when NOT_EQUAL =>
  if T_CATEG /= ATF.DESIRED_TRK_CAT then
  return TRUE;
  else
  return FALSE;
  end if;

end case;

  -- ATOMIC_FILTER based on category-type attributes
  when PLATFORM_IDENTITY_FILTER =>

  -- IDENTITY applies only to platforms below
if ( T_CATEG = SURFACE_PLATFORM ) OR
   ( T_CATEG = SUBSURFACE_PLATFORM ) OR
   ( T_CATEG = AIR_PLATFORM ) then

T_ID := TRACK_IDENTITY ( T );

  case ATF.EQ_REL_ID2 is

  -- IDENTITY_TYPE must be equal to the input parameter value
  -- in order to pass
  when EQUAL =>
    if T_ID = ATF.DESIRED_PLAT_ID then
    return TRUE;
    else
    return FALSE;
    end if;

  -- IDENTITY_TYPE must not be equal to the input parameter value
  -- in order to pass
  when NOT_EQUAL =>
    if T_ID /= ATF.DESIRED_PLAT_ID then
    return TRUE;
    else
    return FALSE;
```

```
      end if;

    end case;

    else -- Non-applicable TRACK types

    -- Since IDENTITY doesn't apply to other TRACKs, if the
    -- ATOMIC_FILTER requires an equality relation to an IDENTITY
    -- it must always fail. Likewise, a non-equal parameter must
    -- always succeed.
  if ATF.EQ_REL_ID2 = EQUAL then
  return FALSE;
  else
  return TRUE;
  end if;

    end if;

  end case;

  end TEST_ATOMIC_FILTER;

  begin -- TEST_FILTER

  -- All TRACKs pass an 'EVERYTHING' FILTER
  if F = EVERYTHING then
  return TRUE;
  else

  -- Get first AND_FILTER
  ANFP := F.FIRST_AND_FILTER;

  -- Test all AND_FILTERs ( if necessary )
  while ANFP /= null loop

   -- Get first ATOMIC_FILTER of this AND_FILTER
  ATFP := ANFP.AND_FLTR.FIRST_ATOMIC_FILTER;

   -- Test all ATOMIC_FILTERs of this AND_FILTER ( if necessary )
  while ATFP /= null loop
```

```
AF := ATFP.ATM_FILTER;

  -- Test the TRACK against this ATOMIC_FILTER
B := TEST_ATOMIC_FILTER ( AF );

  -- A failure of one ATOMIC_FILTER in an AND_FILTER constitutes a
  -- failure of the entire AND_FILTER, so move on to the next
  -- AND_FILTER
if B = FALSE then
exit;
end if;

  -- Get next ATOMIC_FILTER ( previous one passed )
ATFP := ATFP.NEXT_ATOMIC_FILTER;

end loop;

 -- If the TRACK passed all ATOMIC_FILTERs of the previous AND_FILTER,
 -- no need to continue. It passes the FILTER.
if B = TRUE then
return B;
end if;

-- TRACK did not pass the previous AND_FILTER, so get the next one.
ANFP := ANFP.NEXT_AND_FILTER;

end loop;

end if;

return B;

end TEST_FILTER;

--
.................................EVERYTHING................................
function EVERYTHING return FILTER is

F : FILTER;

begin
```

```
return F;
end EVERYTHING;

--
.......................WRITE_FILTER_ARCHIVES_TO_TEXT_FILE...............
procedure WRITE_FILTER_ARCHIVES_TO_TEXT_FILE is

AF : ATOMIC_FILTER;
FC : FILTER_CATEGORY;
TC   : TRACK_CATEGORY;
PID  : IDENTITY_TYPE;
RID : RELATION_ID;
EQ   : EQUALITY_RELATION_ID;
FILTER_FILE : FILTER_INOUT.FILE_TYPE;-- Archive file
FILTER_HIS_FILE : TEXT_IO.FILE_TYPE;-- Text file of all FILTERs
FLTR_NUM : POSITIVE;-- FILTER number in file
F_INDEX : NATURAL;
AND_FLTR_NUM : NATURAL;-- AND_FILTER number in FILTER
ATOMIC_FLTR_OUT : ATOMIC_FILTER_OUT;
WRITE_TIME : ABSOLUTE_TIME;-- Time FILTER archived
FINISHED : BOOLEAN := FALSE;-- Flags when no more FILTERs
DASHES : STRING ( 1.. 80 ) := ( others => '=' );


-- Writes time of archive to text file
procedure PRINT_TIME_OUT is

Y, M, D : NATURAL;
S : FLOAT;

begin

Y := YEAR ( WRITE_TIME );
M := MONTH ( WRITE_TIME );
D := DAY ( WRITE_TIME );
S := TIME_OF_DAY ( WRITE_TIME );

TEXT_IO.PUT ( FILTER_HIS_FILE, NATURAL'IMAGE ( M ) );
TEXT_IO.PUT ( FILTER_HIS_FILE, "/" );
TEXT_IO.PUT ( FILTER_HIS_FILE, NATURAL'IMAGE ( D ) );
TEXT_IO.PUT ( FILTER_HIS_FILE, "/" );
```

```
TEXT_IO.PUT ( FILTER_HIS_FILE, NATURAL'IMAGE (Y - 1900) );
TEXT_IO.PUT ( FILTER_HIS_FILE, " " );
TEXT_IO.PUT ( FILTER_HIS_FILE, NATURAL'IMAGE
     ( HOURS ( TIME_OF_DAY ( WRITE_TIME ) ) ));
TEXT_IO.PUT ( FILTER_HIS_FILE, ':' );
TEXT_IO.PUT ( FILTER_HIS_FILE, NATURAL'IMAGE
     ( MINUTES ( TIME_OF_DAY ( WRITE_TIME ) ) ) );
TEXT_IO.PUT ( FILTER_HIS_FILE, ':' );
TEXT_IO.PUT ( FILTER_HIS_FILE, NATURAL'IMAGE
     ( NATURAL ( SECONDS ( TIME_OF_DAY ( WRITE_TIME ) ) ) ) );
end PRINT_TIME_OUT;

begin -- WRITE_FILTER_ARCHIVES_TO_TEXT_FILE

-- Open archive & create text files
FILTER_INOUT.OPEN ( FILTER_FILE, INOUT_FILE, "FILTER_FILE" );
TEXT_IO.CREATE ( FILTER_HIS_FILE, NAME => "FILTER.HIS" );

-- Read in first archived FILTER
FILTER_INOUT.READ ( FILTER_FILE, ATOMIC_FLTR_OUT );

-- Read in all archived FILTERs and convert them to human-readable
format
-- for output to text file
while NOT FINISHED loop
FLTR_NUM := ATOMIC_FLTR_OUT.FILTER_NUM;
WRITE_TIME := ATOMIC_FLTR_OUT.TIME_OUT;
TEXT_IO.PUT ( FILTER_HIS_FILE, "FILTER NUMBER :" );
TEXT_IO.PUT ( FILTER_HIS_FILE, POSITIVE'IMAGE ( FLTR_NUM ) );
TEXT_IO.SET_COL ( FILTER_HIS_FILE, 35 );
PRINT_TIME_OUT;
TEXT_IO.NEW_LINE ( FILTER_HIS_FILE, 2 );

while ( FLTR_NUM = ATOMIC_FLTR_OUT.FILTER_NUM ) AND ( NOT FINISHED )
loop
AND_FLTR_NUM := ATOMIC_FLTR_OUT.AND_FILTER_NUM;

if AND_FLTR_NUM = 0 then
TEXT_IO.PUT_LINE ( FILTER_HIS_FILE, " ALL TRACKS ACCEPTED" );
TEXT_IO.NEW_LINE ( FILTER_HIS_FILE );
```

```
if NOT FILTER_INOUT.END_OF_FILE ( FILTER_FILE ) then
FILTER_INOUT.READ ( FILTER_FILE, ATOMIC_FLTR_OUT );
else
FINISHED := TRUE;
end if;

else
TEXT_IO.PUT ( FILTER_HIS_FILE, " AND_FILTER NUMBER :" );
TEXT_IO.PUT ( FILTER_HIS_FILE, POSITIVE'IMAGE ( AND_FLTR_NUM ) );
TEXT_IO.NEW_LINE ( FILTER_HIS_FILE );

while ( AND_FLTR_NUM = ATOMIC_FLTR_OUT.AND_FILTER_NUM ) AND
   ( NOT FINISHED ) loop
AF := ATOMIC_FLTR_OUT.ATOMIC_FILTUR;
FC := AF.FILTER_TYPE;
TEXT_IO.SET_COL ( FILTER_HIS_FILE, 7 );

case FC is

when DISTANCE_FILTER =>
RID := AF.D_RELATION;

if AF.D_ATTRIB_ID = RANGE_FROM_REFERENCE_TRACK then
TEXT_IO.PUT ( FILTER_HIS_FILE, "RANGE FROM REFERENCE TRACK" );
TEXT_IO.PUT ( FILTER_HIS_FILE, NATURAL'IMAGE
( TRACK_ID_NUMBER ( AF.REFERENCE_TRACK ) ) );
else
TEXT_IO.PUT ( FILTER_HIS_FILE, "ALTITUDE" );
end if;

case RID is
when EQUAL =>
TEXT_IO.PUT ( FILTER_HIS_FILE, " =" );
when NOT_EQUAL =>
TEXT_IO.PUT ( FILTER_HIS_FILE, " <>" );
when LESS =>
TEXT_IO.PUT ( FILTER_HIS_FILE, " <" );
when LESS_OR_EQUAL =>
TEXT_IO.PUT ( FILTER_HIS_FILE, " <=" );
when GREATER =>
TEXT_IO.PUT ( FILTER_HIS_FILE, " >" );
```

```
when GREATER_OR_EQUAL =>
TEXT_IO.PUT ( FILTER_HIS_FILE, " >=" );
end case;

TEXT_IO.PUT ( FILTER_HIS_FILE, NATURAL'IMAGE ( NATURAL
       ( AF.D_LIMIT ) ) );
TEXT_IO.PUT_LINE ( FILTER_HIS_FILE, " yards" );

when TRACK_CATEGORY_FILTER =>
TC := AF.DESIRED_TRK_CAT;
EQ := AF.EQ_REL_ID1;
TEXT_IO.PUT ( FILTER_HIS_FILE, "TRACK CATEGORY" );

case EQ is
when EQUAL =>
TEXT_IO.PUT ( FILTER_HIS_FILE, " = " );
when NOT_EQUAL =>
TEXT_IO.PUT ( FILTER_HIS_FILE, " <> " );
end case;

case TC is
when TRACK_PKG.UNKNOWN =>
TEXT_IO.PUT ( FILTER_HIS_FILE, "UNKNOWN" );
when SURFACE_PLATFORM =>
TEXT_IO.PUT ( FILTER_HIS_FILE, "SURFACE_PLATFORM" );
when SUBSURFACE_PLATFORM =>
TEXT_IO.PUT ( FILTER_HIS_FILE, "SUBSURFACE_PLATFORM" );
when AIR_PLATFORM =>
TEXT_IO.PUT ( FILTER_HIS_FILE, "AIR_PLATFORM" );
when REGION =>
TEXT_IO.PUT ( FILTER_HIS_FILE, "REGION" );
when SPECIAL_POINT =>
TEXT_IO.PUT ( FILTER_HIS_FILE, "SPECIAL_POINT" );
when PATH =>
TEXT_IO.PUT ( FILTER_HIS_FILE, "PATH" );
when MAN_IN_WATER =>
TEXT_IO.PUT ( FILTER_HIS_FILE, "MAN_IN_WATER" );
when NON_DISPLAYABLE =>
TEXT_IO.PUT ( FILTER_HIS_FILE, "NON_DISPLAYABLE" );
end case;
```

```
TEXT_IO.NEW_LINE ( FILTER_HIS_FILE );

when PLATFORM_IDENTITY_FILTER =>
PID := AF.DESIRED_PLAT_ID;
EQ := AF.EQ_REL_ID2;
TEXT_IO.PUT ( FILTER_HIS_FILE, "PLATFORM IDENTITY" );

case EQ is
when EQUAL =>
TEXT_IO.PUT ( FILTER_HIS_FILE, " = " );
when NOT_EQUAL =>
TEXT_IO.PUT ( FILTER_HIS_FILE, " <> " );
end case;

case PID is
when TRACK_PKG.UNKNOWN =>
TEXT_IO.PUT ( FILTER_HIS_FILE, "UNKNOWN" );
when FRIENDLY =>
TEXT_IO.PUT ( FILTER_HIS_FILE, "FRIENDLY" );
when HOSTILE =>
TEXT_IO.PUT ( FILTER_HIS_FILE, "HOSTILE" );
when NEUTRAL =>
TEXT_IO.PUT ( FILTER_HIS_FILE, "NEUTRAL" );
end case;

TEXT_IO.NEW_LINE ( FILTER_HIS_FILE );

end case;

if NOT FILTER_INOUT.END_OF_FILE ( FILTER_FILE ) then
FILTER_INOUT.READ ( FILTER_FILE, ATOMIC_FLTR_OUT );
else
FINISHED := TRUE;
end if;

end loop;

TEXT_IO.NEW_LINE ( FILTER_HIS_FILE );

end if;
```

```
 end loop;

 TEXT_IO.PUT_LINE ( FILTER_HIS_FILE, DASHES );

 end loop;

 FILTER_INOUT.CLOSE ( FILTER_FILE );
 TEXT_IO.CLOSE ( FILTER_HIS_FILE );

 end WRITE_FILTER_ARCHIVES_TO_TEXT_FILE;

--
........................................................................

end FILTER_PKG;
```

APPENDIX E

CPA PACKAGE

```
--
==========================================================================
--
-- Authors : Richard T. Irwin
-- Willie K. Bolick
--
-- Date : 29 August 1991
--
--
..........................................................................
-- Description : Defines data type CPA_TYPE and associated function
FIND_CPA
--
--========================================================================

with VECTOR_2_PKG, ABSOLUTE_TIME_PKG, TRACK_PKG;

use VECTOR_2_PKG, ABSOLUTE_TIME_PKG, TRACK_PKG;

package CPA_PKG is

 type CPA_TYPE is
 record
 CPA_BEARING_AND_RANGE : VECTOR_2; -- Bearing & range to target from
             -- reference at CPA
 TIME_OF_CPA : ABSOLUTE_TIME; -- Time when CPA occurs
 end record;

 -- Finds Closest Point of Approach of target track to the reference
track
 function FIND_CPA
 ( TARGET_TRK, REFERENCE_TRACK : TRACK ) return CPA_TYPE;

 pragma INLINE ( FIND_CPA );

end CPA_PKG;
```

```
with ANGLE_PKG, SPEED_PKG, DISTANCE_PKG, GLOBAL_POSITION_PKG,
RELATIVE_TIME_PKG,
 VELOCITY_PKG, RELATIVE_POSITION_PKG, MATH;

use ANGLE_PKG, SPEED_PKG, DISTANCE_PKG, GLOBAL_POSITION_PKG,
RELATIVE_TIME_PKG,
 VELOCITY_PKG, RELATIVE_POSITION_PKG;

package body CPA_PKG is

 function FIND_CPA
 ( TARGET_TRK, REFERENCE_TRACK : TRACK ) return CPA_TYPE is

 CPA_TO_TARGET : CPA_TYPE;
 TGT_BEARING : ANGLE;-- true brg to target
 TGT_RANGE : DISTANCE;-- range to target (yds)
 TGT_REL_SPEED : SPEED;-- rel spd of target
 TGT_REL_COURSE : ANGLE;-- rel crs of target
 PERPENDICULAR_1,-- perp of tgt rel crs
 PERPENDICULAR_2 : ANGLE;-- perp of tgt rel crs
 P1_DIFF, -- diff bet tgt rel crs
 P2_DIFF : ANGLE;-- & the perpendiculars
 CPA_BEARING : ANGLE;-- bearing to target at cpa
 CPA_RANGE : DISTANCE;-- range to target at cpa
 CPA_TIME : RELATIVE_TIME;-- time in secs to cpa
 ALPHA : ANGLE;-- angle bet bearing to
           -- tgt & bearing to cpa
 BRAVO : ANGLE;-- angle bet bearing to
           -- tgt & tgt rel crs
 REL_VELOCITY : VELOCITY;
 LAST_TGT_POSITION,
 LAST_REF_POSITION,
 OPENING_POS_TGT,
 OPENING_POS_REF : GLOBAL_POSITION;
 OPENING_RG : DISTANCE;
 OBS_TIME : ABSOLUTE_TIME := NOW;

 begin
```

```
-- Get current positions of target & reference tracks
LAST_REF_POSITION := CURRENT_POSITION ( REFERENCE_TRACK );
LAST_TGT_POSITION := CURRENT_POSITION ( TARGET_TRK );

-- Find present bearing & range to target
TGT_BEARING := BEARING_TO ( FIND_RELATIVE_POSITION
     ( LAST_TGT_POSITION, LAST_REF_POSITION ) );
TGT_RANGE := RANGE_OF ( FIND_RELATIVE_POSITION
       ( LAST_TGT_POSITION, LAST_REF_POSITION ));

-- Get target's relative course & speed
REL_VELOCITY := TARGET_RELATIVE_VELOCITY ( REFERENCE_TRACK, TARGET_TRK
);
TGT_REL_COURSE := COURSE ( REL_VELOCITY );
TGT_REL_SPEED := SPD ( REL_VELOCITY );

-- Get target's & reference's position again to determine if they
-- are opening one another
OPENING_POS_REF := CURRENT_POSITION ( REFERENCE_TRACK );
OPENING_POS_TGT := CURRENT_POSITION ( TARGET_TRK );
OPENING_RG := RANGE_OF ( FIND_RELATIVE_POSITION
       ( OPENING_POS_TGT, OPENING_POS_REF ) );

-- If target & reference are opening or if the target has no relative
speed,
-- no CPA possible
if ( OPENING_RG > TGT_RANGE ) or ( TGT_REL_SPEED = 0.0 ) then

CPA_BEARING := TGT_BEARING;
CPA_RANGE := TGT_RANGE;
CPA_TIME := 0.0;

else

-- The bearing to the target at cpa will be 90 degrees +/- the target's
-- relative course. The problem is finding out which one applies. To
-- determine the correct one, computations are made on both
perpendiculars
-- The perpendicular closest to the target's bearing is the cpa bearing.

-- Subtract 90 degrees from target's relative course to get perp1
PERPENDICULAR_1 := TGT_REL_COURSE - MATH.PI / 2.0;
```

```
-- If target's relative course < 270, add 90 degrees to get perp2,
-- otherwise subtract 90 degrees
if TGT_REL_COURSE < MATH.PI * 3.0 / 2.0 then
PERPENDICULAR_2 := TGT_REL_COURSE + MATH.PI / 2.0;
else
PERPENDICULAR_2 := PERPENDICULAR_1 - MATH.PI;
end if;

-- If computed perp1 is negative, add 360 degrees to correct
if PERPENDICULAR_1 < 0.0 then
PERPENDICULAR_1 := MATH.PI * 2.0 + PERPENDICULAR_1;
end if;

-- If computed perp2 is negative, add 360 degrees to correct
if PERPENDICULAR_2 < 0.0 then
PERPENDICULAR_2 := MATH.PI * 2.0 + PERPENDICULAR_2;
end if;

-- Compute absolute difference between target's bearing & perp1
P1_DIFF := ABS ( TGT_BEARING - PERPENDICULAR_1 );

-- If difference is > 180 degrees in one direction, it is < 180 in
-- the other direction, so choose the shortest one
if P1_DIFF > MATH.PI then
P1_DIFF := MATH.PI * 2.0 - P1_DIFF;
end if;

-- Compute absolute difference between target's bearing & perp2
P2_DIFF := ABS ( TGT_BEARING - PERPENDICULAR_2 );

-- If difference is > 180 degrees in one direction, it is < 180 in
-- the other direction, so choose the shortest one
if P2_DIFF > MATH.PI then
P2_DIFF := MATH.PI * 2.0 - P2_DIFF;
end if;

-- The smallest difference determines the correct perpendicular to use
-- as cpa bearing
if P1_DIFF < P2_DIFF then
CPA_BEARING := PERPENDICULAR_1;
```

```
elsif P1_DIFF > P2_DIFF then
CPA_BEARING := PERPENDICULAR_2;
else
-- ** CBDR ** ( Constant Bearing, Decreasing Range ) Crash coming!
CPA_BEARING := TGT_BEARING;
end if;

-- Need to find angle between cpa bearing and target's current bearing
-- so we can compute the distance from target's current position and
-- its position at cpa
ALPHA := ABS ( CPA_BEARING - TGT_BEARING );

-- If the angle is > 180 degrees in one direction, it is < 180 in
-- the other direction, so choose the shortest one
if ALPHA > MATH.PI then
ALPHA := MATH.PI * 2.0 - ALPHA;
end if;

-- The angle between the target's relative course and its bearing at cpa
-- is 90 degrees. We just computed a second angle ( ALPHA ) of the
-- triangle, so the remaining angle of the triangle is 90 degrees minus
-- ALPHA. This angle ( BRAVO ) gives us the angle between the target's
-- relative course and the true bearing to the target.
BRAVO := MATH.PI / 2.0 - ALPHA;

-- Compute range to target at cpa and time of cpa
if ALPHA = 0.0 then -- ** CBDR **
CPA_TIME := TGT_RANGE / TGT_REL_SPEED;
CPA_RANGE := 0.0;
else
CPA_RANGE := TGT_RANGE * DISTANCE ( SIN ( BRAVO ) );

-- Pythagorean Theorem used
CPA_TIME := SQRT ( TGT_RANGE * TGT_RANGE - CPA_RANGE * CPA_RANGE ) /
       TGT_REL_SPEED;
end if;

end if;

CPA_TO_TARGET.CPA_BEARING_AND_RANGE := MAKE_POLAR_VECTOR_2
( CPA_RANGE, CPA_BEARING );
```

```
 CPA_TO_TARGET.TIME_OF_CPA := CPA_TIME + OBS_TIME;

 return CPA_TO_TARGET;

 end FIND_CPA;

end CPA_PKG;
```

APPENDIX F

VELOCITY PACKAGE

```
--
==========================================================================
--
-- Authors : Richard T. Irwin
--  Willie K. Bolick
--
-- Date : 29 August 1991
--
--
..........................................................................
--
-- Description : Defines data subtype VELOCITY and associated functions
--
--
==========================================================================

with VECTOR_2_PKG, SPEED_PKG, ANGLE_PKG;

use VECTOR_2_PKG, SPEED_PKG, ANGLE_PKG;

package VELOCITY_PKG is

 subtype VELOCITY is VECTOR_2;-- Course and speed vector

 -- Returns course & speed vector, given course & speed values
 function MAKE_VELOCITY
 ( SPD : SPEED;
 COURSE : ANGLE ) return VELOCITY renames
VECTOR_2_PKG.MAKE_POLAR_VECTOR_2;

 -- Returns course attribute of a velocity vector
 function COURSE
 ( V : VELOCITY ) return ANGLE renames VECTOR_2_PKG.DIRECTION;

 -- Returns speed attribute of a velocity vector
 function SPD
```

```
 ( V : VELOCITY ) return SPEED renames VECTOR_2_PKG.LENGTH;

 pragma INLINE ( MAKE_VELOCITY, COURSE, SPD );

end VELOCITY_PKG;
```

APPENDIX G

VECTOR 2 PACKAGE

```
--
==========================================================================
--
-- Authors : Richard T. Irwin
--  Willie K. Bolick
--
-- Date : 29 August 1991
--
--
..........................................................................
--
-- Description : Defines abstract data type VECTOR_2 and associated
functions
--
--
==========================================================================

with ANGLE_PKG, MATH;

use ANGLE_PKG;

package VECTOR_2_PKG is

 type VECTOR_2 is private;

 function SQRT ( F : FLOAT ) return FLOAT renames MATH.SQRT;

 -- Returns a vector, given a length and an angle in radians
 function MAKE_POLAR_VECTOR_2
 ( LENGTH : FLOAT;
 DIRECTION : ANGLE ) return VECTOR_2;

 -- Returns the length attribute of a given VECTOR_2
 function LENGTH
 ( V : VECTOR_2 ) return FLOAT;
```

```
-- Returns the angle attribute of a given VECTOR_2
function DIRECTION
( V : VECTOR_2 ) return ANGLE;

-- Returns a vector, given its end point in terms of X & Y coordinates
function MAKE_CARTESIAN_VECTOR_2
( X, Y : FLOAT ) return VECTOR_2;

-- Returns the X-coordinate of a vector
function X_COORDINATE
( V : VECTOR_2 ) return FLOAT;

-- Returns the Y-coordinate of a vector
function Y_COORDINATE
( V : VECTOR_2 ) return FLOAT;

-- Returns the resultant sum of 2 vectors
function "+"
( V1, V2 : VECTOR_2 ) return VECTOR_2;

-- Returns the resultant difference of 2 vectors
function "-"
( V1, V2 : VECTOR_2 ) return VECTOR_2;

-- Returns the resultant dot product of 2 vectors
function DOT_PRODUCT
( V1, V2 : VECTOR_2 ) return FLOAT;

-- Returns the resultant product of a vector and a scale factor
function "*"
( V : VECTOR_2;
SCALE_FACTOR : FLOAT ) return VECTOR_2;

-- Returns a vector rotated about a given angle
function ROTATE
( V : VECTOR_2;
A : ANGLE ) return VECTOR_2;

-- Returns a normalized vector
function NORMALIZE
( V : VECTOR_2 ) return VECTOR_2;
```

```
 pragma INLINE
 ( MAKE_POLAR_VECTOR_2, LENGTH, DIRECTION, MAKE_CARTESIAN_VECTOR_2,
 X_COORDINATE, Y_COORDINATE, "+", "-", DOT_PRODUCT, ROTATE, NORMALIZE );

private

 type VECTOR_2 is
 record
 X, Y : FLOAT;
 end record;

 ZERO : constant VECTOR_2 := ( 0.0, 0.0 );

end VECTOR_2_PKG;

--
=========================================================================
--
-- Authors : Richard T. Irwin
--  Willie K. Bolick
--
-- Date : 29 August 1991
--
--
=========================================================================

package body VECTOR_2_PKG is

--
..............................MAKE_POLAR_VECTOR_2.....................
 function MAKE_POLAR_VECTOR_2
 ( LENGTH : FLOAT;
 DIRECTION : ANGLE ) return VECTOR_2 is

 V : VECTOR_2;

 begin

 V.X := LENGTH * SIN ( DIRECTION );
 V.Y := LENGTH * COS ( DIRECTION );
 return V;
```

```
end MAKE_POLAR_VECTOR_2;

--
..................................LENGTH...............................
function LENGTH
( V : VECTOR_2 ) return FLOAT is

begin
return SQRT ( V.X * V.X + V.Y * V.Y );
end LENGTH;

--
...............................DIRECTION...............................
function DIRECTION
( V : VECTOR_2 ) return ANGLE is

X, Y : FLOAT;
A : ANGLE;

begin

X := V.X;
Y := V.Y;

if X = 0.0 then

if Y >= 0.0 then
return DEGREES_TO_RADIANS ( 0.0 );
else
return DEGREES_TO_RADIANS ( 180.0 );
end if;

elsif Y / X < 0.0 then -- Either X or Y is negative

if Y < 0.0 then -- Y is negative
return DEGREES_TO_RADIANS ( 90.0 ) - ARCTAN ( Y / X );
else -- X is negative
return DEGREES_TO_RADIANS ( 270.0 ) - ARCTAN ( Y / X );
end if;
```

```
else

if X < 0.0 then -- X and Y are both negative
return DEGREES_TO_RADIANS ( 270.0 ) - ARCTAN ( Y / X );
else -- X and Y are both positive ( Y could be 0.0 )
return DEGREES_TO_RADIANS ( 90.0 ) - ARCTAN ( Y / X );
end if;

end if;

end DIRECTION;

--
...........................MAKE_CARTESIAN_VECTOR_2....................
function MAKE_CARTESIAN_VECTOR_2
( X, Y : FLOAT ) return VECTOR_2 is

V : VECTOR_2;

begin

V.X := X;
V.Y := Y;
return V;

end MAKE_CARTESIAN_VECTOR_2;

--
................................X_COORDINATE..........................
function X_COORDINATE
( V : VECTOR_2 ) return FLOAT is

begin
return V.X;
end X_COORDINATE;

--
................................Y_COORDINATE..........................
function Y_COORDINATE
( V : VECTOR_2 ) return FLOAT is

begin
```

```
return V.Y;
end Y_COORDINATE;

--
................................."+"...............................
function "+"
( V1, V2 : VECTOR_2 ) return VECTOR_2 is

V : VECTOR_2;

begin

V.X := V1.X + V2.X;
V.Y := V1.Y + V2.Y;
return V;

end "+";

--................................."-
"..................................
function "-"
( V1, V2 : VECTOR_2 ) return VECTOR_2 is

V : VECTOR_2;

begin

V.X := V1.X - V2.X;
V.Y := V1.Y - V2.Y;
return V;

end "-";

--
.............................DOT_PRODUCT.........................
function DOT_PRODUCT
( V1, V2 : VECTOR_2 ) return FLOAT is

begin
return V1.X * V2.X + V1.Y * V2.Y;
end DOT_PRODUCT;
```

```
--
...................................."*"................................
function "*"
( V : VECTOR_2;
SCALE_FACTOR : FLOAT ) return VECTOR_2 is

V2 : VECTOR_2;

begin

-- Length ( result ) = length ( v ) * scale_factor
-- Direction ( result ) = direction ( v )

V2.X := V.X * SCALE_FACTOR;
V2.Y := V.Y * SCALE_FACTOR;
return V2;

end "*";

--
.................................ROTATE................................
function ROTATE
( V : VECTOR_2;
A : ANGLE ) return VECTOR_2 is

D : ANGLE;
V2 : VECTOR_2;

begin

-- Direction ( result ) = direction ( v ) + a
-- Length ( result ) = length ( v )

D := DIRECTION ( V ) + A;
V2.X := LENGTH ( V ) * SIN ( D );
V2.Y := LENGTH ( V ) * COS ( D );
return V2;

end ROTATE;
```

```
--
...............................NORMALIZE..............................
 function NORMALIZE
 ( V : VECTOR_2 ) return VECTOR_2 is

 D : ANGLE;
 V2 : VECTOR_2;

 begin

 -- Direction ( result ) = direction ( v )
 -- Length ( result ) = 1.0

 D := DIRECTION ( V );
 V2.X := COS ( D );
 V2.Y := SIN ( D );
 return V2;

 end NORMALIZE;

--
......................................................................

end VECTOR_2_PKG;
```

APPENDIX H

VECTOR 3 PACKAGE

```
--
==========================================================================
--
-- Authors : Richard T. Irwin
--  Willie K. Bolick
--
-- Date : 29 August 1991
--
--
..........................................................................
--
-- Description : Defines abstract data type VECTOR_3 and associated
functions
--
--
==========================================================================

with ANGLE_PKG, MATH;

use ANGLE_PKG;

package VECTOR_3_PKG is

 type VECTOR_3 is private;

 function SQRT ( F : FLOAT ) return FLOAT renames MATH.SQRT;

 -- Returns a vector, given a length, an angle in radians, and an azimuth
 -- in radians
 function MAKE_POLAR_VECTOR_3
 ( LENGTH : FLOAT;
 THETA : ANGLE;
 PHI : AZIMUTH ) return VECTOR_3;

 -- Returns the length attribute of a given VECTOR_2
 function LENGTH
```

```
( V : VECTOR_3 ) return FLOAT;

function THETA
( V : VECTOR_3 ) return ANGLE;

function PHI
( V : VECTOR_3 ) return AZIMUTH;

-- Returns a vector, given its end point in terms of X, Y, & Z
coordinates
function MAKE_CARTESIAN_VECTOR_3
( X, Y, Z : FLOAT ) return VECTOR_3;

-- Returns the X-coordinate of a vector
function X_COORDINATE
( V : VECTOR_3 ) return FLOAT;

-- Returns the Y-coordinate of a vector
function Y_COORDINATE
( V : VECTOR_3 ) return FLOAT;

-- Returns the Z-coordinate of a vector
function Z_COORDINATE
( V : VECTOR_3 ) return FLOAT;

-- Returns the resultant sum of 2 vectors
function "+"
( V1, V2 : VECTOR_3 ) return VECTOR_3;

-- Returns the resultant difference of 2 vectors
function "-"
( V1, V2 : VECTOR_3 ) return VECTOR_3;

-- Returns the resultant dot product of 2 vectors
function DOT_PRODUCT
( V1, V2 : VECTOR_3 ) return FLOAT;

function CROSS_PRODUCT
( V1, V2 : VECTOR_3 ) return VECTOR_3;

-- Length ( result ) = length ( v ) * scale_factor
```

```
 function SCALE
 ( V : VECTOR_3;
 SCALE_FACTOR : FLOAT ) return VECTOR_3;


 -- Returns a normalized vector
 -- length ( result ) = 1.0
 function NORMALIZE
 ( V : VECTOR_3 ) return VECTOR_3;


 pragma INLINE ( MAKE_POLAR_VECTOR_3, LENGTH, THETA, PHI,
      MAKE_CARTESIAN_VECTOR_3, X_COORDINATE, Y_COORDINATE,
      Z_COORDINATE, "+", "-", DOT_PRODUCT, CROSS_PRODUCT, SCALE,
      NORMALIZE );

private

 type VECTOR_3 is
 record
 X, Y, Z : FLOAT;
 end record;

 ZERO : constant VECTOR_3 := ( 0.0, 0.0, 0.0 );

end VECTOR_3_PKG;

--
=========================================================================
-- Authors : Richard T. Irwin
--  Willie K. Bolick
--
-- Date : 29 August 1991
--
--
==========================================================================

package body VECTOR_3_PKG is-
..............................MAKE_POLAR_VECTOR_3.....................
 function MAKE_POLAR_VECTOR_3
 ( LENGTH : FLOAT;
 THETA : ANGLE;
```

```
PHI : AZIMUTH ) return VECTOR_3 is

V : VECTOR_3;
R : FLOAT;

begin

R := LENGTH * COS ( PHI );
V.X := R * COS ( THETA );
V.Y := R * SIN ( THETA );
V.Z := LENGTH * SIN ( PHI );
return V;

end MAKE_POLAR_VECTOR_3;

--
..................................LENGTH..............................
function LENGTH
( V : VECTOR_3 ) return FLOAT is

R : FLOAT;

begin

R := SQRT ( V.X * V.X + V.Y * V.Y );
return SQRT ( R * R + V.Z * V.Z );

end LENGTH;

--
..................................THETA...............................
function THETA
( V : VECTOR_3 ) return ANGLE is

begin
return ARCTAN ( V.Y / V.X );
end THETA;
-..................................PHI.................................
function PHI
( V : VECTOR_3 ) return AZIMUTH is
R : FLOAT;
```

```
begin

R := SQRT ( V.X * V.X + V.Y * V.Y );
return AZIMUTH ( ARCTAN ( V.Z / R ) );

end PHI;

--
............................MAKE_CARTESIAN_VECTOR_3.....................
function MAKE_CARTESIAN_VECTOR_3
( X, Y, Z : FLOAT ) return VECTOR_3 is

V : VECTOR_3;

begin

V.X := X;
V.Y := Y;
V.Z := Z;
return V;

end MAKE_CARTESIAN_VECTOR_3;

--
.................................X_COORDINATE..........................
function X_COORDINATE
( V : VECTOR_3 ) return FLOAT is

begin
return V.X;
end X_COORDINATE;

--
................................Y_COORDINATE...........................
function Y_COORDINATE
( V : VECTOR_3 ) return FLOAT is

begin
return V.Y;
end Y_COORDINATE;
```

```
--
...............................Z_COORDINATE...........................
 function Z_COORDINATE
 ( V : VECTOR_3 ) return FLOAT is

 begin
 return V.Z;
 end Z_COORDINATE;

--
..................................."+"...................................
 function "+"
 ( V1, V2 : VECTOR_3 ) return VECTOR_3 is

 V : VECTOR_3;

 begin

 V.X := V1.X + V2.X;
 V.Y := V1.Y + V2.Y;
 V.Z := V1.Z + V2.Z;
 return V;

 end "+";

--..................................."-
"......................................
 function "-"
 ( V1, V2 : VECTOR_3 ) return VECTOR_3 is

 V : VECTOR_3;

 begin

 V.X := V1.X - V2.X;
 V.Y := V1.Y - V2.Y;
 V.Z := V1.Z - V2.Z;
 return V;

 end "-";
```

```
--
................................DOT_PRODUCT.............................
function DOT_PRODUCT
( V1, V2 : VECTOR_3 ) return FLOAT is

begin
return V1.X * V2.X + V1.Y * V2.Y + V1.Z * V2.Z;
end DOT_PRODUCT;

--
................................CROSS_PRODUCT...........................
function CROSS_PRODUCT
( V1, V2 : VECTOR_3 ) return VECTOR_3 is

V : VECTOR_3;

begin

V.X := V1.Y * V2.Z - V1.Z * V2.Y;
V.Y := V1.Z * V2.X - V1.X * V2.Z;
V.Z := V1.X * V2.Y - V1.Y * V2.X;
return V;

end CROSS_PRODUCT;
--
...................................SCALE................................
function SCALE
( V : VECTOR_3;
SCALE_FACTOR : FLOAT ) return VECTOR_3 is

V3 : VECTOR_3;

begin

-- length ( result ) = length ( v ) * scale_factor

V3.X := V.X * SCALE_FACTOR;
V3.Y := V.Y * SCALE_FACTOR;
V3.Z := V.Z * SCALE_FACTOR;
return V3;

end SCALE;
```

```
--
..................................NORMALIZE..............................
 function NORMALIZE
 ( V : VECTOR_3 ) return VECTOR_3 is

 R : FLOAT;
 PHI : AZIMUTH;
 THETA : ANGLE;
 V3 : VECTOR_3;

 begin

 -- length ( result ) = 1.0

 THETA := ARCTAN ( V.Y / V.X );
 R := SQRT ( V.X * V.X + V.Y * V.Y );
 PHI := AZIMUTH ( ARCTAN ( V.Z / R ) );
 V3.Z := SIN ( ANGLE ( PHI ) );
 R := COS ( ANGLE ( PHI ) );
 V3.Y := R * SIN ( THETA );
 V3.X := R * COS ( THETA );
 return V3;

 end NORMALIZE;

--
.........................................................................
......
end VECTOR_3_PKG;
```

APPENDIX I

SPEED PACKAGE

```
--
==========================================================================
--
-- Authors : Richard T. Irwin
--  Willie K. Bolick
--
-- Date : 29 August 1991
--
--
..........................................................................
--
-- Description : Defines data type SPEED and associated functions
--
--
==========================================================================

with MATH;

use MATH;

package SPEED_PKG is

 subtype SPEED is FLOAT; -- Units : yards per second

 -- Returns yards per second, given knots ( nautical miles per hour )
 function MAKE_SPEED
 ( KNOTS : FLOAT ) return SPEED;

 -- Returns knots, given yards per second
 function SPEED_IN_KNOTS
 ( S : SPEED ) return FLOAT;

 pragma INLINE
 ( MAKE_SPEED, SPEED_IN_KNOTS );

end SPEED_PKG;
```

```
--
=========================================================================
--
-- Authors : Richard T. Irwin
--  Willie K. Bolick
--
-- Date : 29 August 1991
--
--
=========================================================================

package body SPEED_PKG is

 YDS_IN_KNOT : constant FLOAT := 6080.2 / 3.0;
 SECONDS_IN_HOUR : constant FLOAT := 3600.0;

--
..............................MAKE_SPEED..............................
 function MAKE_SPEED
 ( KNOTS : FLOAT ) return SPEED is
 begin
 return ( KNOTS * YDS_IN_KNOT ) / SECONDS_IN_HOUR;
 end MAKE_SPEED;

--
............................SPEED_IN_KNOTS............................
 function SPEED_IN_KNOTS
 ( S : SPEED ) return FLOAT is
 begin
 return ( S * SECONDS_IN_HOUR ) / YDS_IN_KNOT;
 end SPEED_IN_KNOTS;

--
......................................................................
......

end SPEED_PKG;
```

APPENDIX J

ANGLE PACKAGE

```
--
==========================================================================
--
-- Authors : Richard T. Irwin
--  Willie K. Bolick
--
-- Date : 29 August 1991
--
--
..........................................................................
--
-- Description : Defines data subtypes ANGLE, AZIMUTH, and associated
--  functions
--
--
==========================================================================

with MATH;

use MATH;

package ANGLE_PKG is

 subtype ANGLE is
 FLOAT range -2.0 * PI .. 2.0 * PI; -- Units of radians
 subtype AZIMUTH is
 ANGLE range -1.0 * PI .. PI; -- Units of radians

 function DEGREES_TO_RADIANS ( X : FLOAT ) return ANGLE;
 -- Converts compass degree value to its equivalent radian value

 function RADIANS_TO_DEGREES ( A : ANGLE ) return FLOAT;
 -- Converts radian value to its equivalent compass degree value

 function SIN ( A : ANGLE ) return FLOAT renames MATH.SIN;
 function COS ( A : ANGLE ) return FLOAT renames MATH.COS;
```

```
 function ARCTAN ( A : ANGLE ) return FLOAT renames MATH.ARCTAN;
 function ARCSIN ( A : ANGLE ) return FLOAT renames MATH.ARCSIN;

 pragma INLINE ( DEGREES_TO_RADIANS, RADIANS_TO_DEGREES, SIN, COS,
ARCTAN,
      ARCSIN );

end ANGLE_PKG;

--
=======================================================================
--
-- Authors : Richard T. Irwin
--  Willie K. Bolick
--
-- Date : 29 August 1991
--
--
=======================================================================
package body ANGLE_PKG is

 CONVERSION_FACTOR : constant FLOAT := 180.0 / PI;

--
.............................DEGREES_TO_RADIANS........................
 function DEGREES_TO_RADIANS ( X : FLOAT ) return ANGLE is
 begin
 return ANGLE ( X / CONVERSION_FACTOR );
 end DEGREES_TO_RADIANS;

--
.............................RADIANS_TO_DEGREES........................
 function RADIANS_TO_DEGREES ( A : ANGLE ) return FLOAT is
 F : FLOAT;
 begin
 F := FLOAT ( A ) * CONVERSION_FACTOR;
 if F < 0.0 then
 return 360.0 + F;
 end if;
 return F;
 end
RADIANS_TO_DEGREES;...................................................
end ANGLE_PKG;
```

APPENDIX K

ABSOLUTE TIME PACKAGE

```
--
========================================================================
--
-- Authors : Richard T. Irwin
--  Willie K. Bolick
--
-- Date : 29 August 1991
--
--
........................................................................
--
-- Description : Defines abstract data type ABSOLUTE_TIME and associated
--  functions
--
--
========================================================================

with RELATIVE_TIME_PKG;

use RELATIVE_TIME_PKG;

package ABSOLUTE_TIME_PKG is

 type ABSOLUTE_TIME is private;

 function NOW return ABSOLUTE_TIME;
 -- Converts CALENDAR.CLOCK time to ABSOLUTE_TIME

 function MAKE_ABSOLUTE_TIME
 ( YEAR, MONTH, DAY : NATURAL;
 TIME_OF_DAY : RELATIVE_TIME ) return ABSOLUTE_TIME;
 -- Accepts numerical values of year, month, day, and the time of day
 -- ( represented in seconds ). Converts inputted values to ABSOLUTE_TIME

 function YEAR
 ( T : ABSOLUTE_TIME ) return NATURAL;
```

```
-- Returns the value of the year contained in the ABSOLUTE_TIME input

function MONTH
( T : ABSOLUTE_TIME ) return NATURAL;
-- Returns the value of the month contained in the ABSOLUTE_TIME input

function DAY
( T : ABSOLUTE_TIME ) return NATURAL;
-- Returns the value of the day contained in the ABSOLUTE_TIME input

function TIME_OF_DAY
( T : ABSOLUTE_TIME ) return RELATIVE_TIME;
-- Returns the value of the time of day ( in seconds ) contained in the
-- ABSOLUTE_TIME input

function "+"
( ABT : ABSOLUTE_TIME;
RT : RELATIVE_TIME ) return ABSOLUTE_TIME;

function "+"
( RT : RELATIVE_TIME;
ABT : ABSOLUTE_TIME ) return ABSOLUTE_TIME;

function "-"
( T1, T2 : ABSOLUTE_TIME ) return RELATIVE_TIME;

function "<"
( T1, T2 : ABSOLUTE_TIME ) return BOOLEAN;

pragma INLINE
( MAKE_ABSOLUTE_TIME, YEAR, MONTH, DAY, TIME_OF_DAY );

private

type ABSOLUTE_TIME is
record
ABS_YEAR : NATURAL;
ABS_MONTH : NATURAL;
ABS_DAY : NATURAL;
ABS_HOUR : NATURAL;
ABS_MINUTE : NATURAL;
```

```
 ABS_SECONDS : FLOAT;
 end record;

 BEGINNING : constant ABSOLUTE_TIME := NOW;

end ABSOLUTE_TIME_PKG;

--
========================================================================
--
-- Authors : Richard T. Irwin
--  Willie K. Bolick
--
-- Date : 29 August 1991
--
--
========================================================================

with CALENDAR;

use CALENDAR;

package body ABSOLUTE_TIME_PKG is

--
...................................NOW...................................
 function NOW return ABSOLUTE_TIME is

 ABT : ABSOLUTE_TIME;
 SEC : DAY_DURATION;
 CT : TIME;

 begin

 CT := CLOCK; -- Get system time clock value now
 SEC := SECONDS ( CT ); -- Convert time to seconds

 ABT.ABS_YEAR := NATURAL ( YEAR ( CT ) );
 ABT.ABS_MONTH := NATURAL ( MONTH ( CT ) );
 ABT.ABS_DAY := NATURAL ( DAY ( CT ) );
 ABT.ABS_HOUR := NATURAL ( FLOAT ( SEC ) ) / 3600;
 ABT.ABS_MINUTE := NATURAL ( FLOAT ( SEC ) -
```

```
FLOAT ( ABT.ABS_HOUR * 3600 ) ) / 60;
ABT.ABS_SECONDS := FLOAT ( SEC ) - FLOAT ( ( ABT.ABS_HOUR * 3600 ) +
( ABT.ABS_MINUTE * 60 ) );

return ABT;

end NOW;

--
.............................MAKE_ABSOLUTE_TIME.........................
function MAKE_ABSOLUTE_TIME
( YEAR, MONTH, DAY : NATURAL;
TIME_OF_DAY : RELATIVE_TIME ) return ABSOLUTE_TIME is

ABT : ABSOLUTE_TIME;

begin

ABT.ABS_YEAR := YEAR;
ABT.ABS_MONTH := MONTH;
ABT.ABS_DAY := DAY;
ABT.ABS_HOUR := NATURAL ( TIME_OF_DAY ) / 3600;
ABT.ABS_MINUTE := ( NATURAL ( TIME_OF_DAY ) -
ABT.ABS_HOUR * 3600 ) / 60;
ABT.ABS_SECONDS := FLOAT ( TIME_OF_DAY ) -
FLOAT ( ( ABT.ABS_HOUR * 3600 ) +
( ABT.ABS_MINUTE * 60 ) );

return ABT;

end MAKE_ABSOLUTE_TIME;

--
...................................YEAR.................................
function YEAR
( T : ABSOLUTE_TIME ) return NATURAL is

begin
return T.ABS_YEAR;
end YEAR;
```

```
--
...................................MONTH..................................
 function MONTH
 ( T : ABSOLUTE_TIME ) return NATURAL is

 begin
 return T.ABS_MONTH;
 end MONTH;

--
....................................DAY...................................
 function DAY
 ( T : ABSOLUTE_TIME ) return NATURAL is

 begin
 return T.ABS_DAY;
 end DAY;

--
...............................TIME_OF_DAY................................
 function TIME_OF_DAY
 ( T : ABSOLUTE_TIME ) return RELATIVE_TIME is

 RT : RELATIVE_TIME;

 begin

 RT := RELATIVE_TIME ( T.ABS_HOUR * 3600 + T.ABS_MINUTE * 60 ) +
 RELATIVE_TIME ( T.ABS_SECONDS );

 return RT;

 end TIME_OF_DAY;

--
...................................."+"...................................
 function "+"
 ( ABT : ABSOLUTE_TIME;
 RT : RELATIVE_TIME ) return ABSOLUTE_TIME is

 RABT : ABSOLUTE_TIME;
 RTM : RELATIVE_TIME;
```

```
TM : TIME;
Y : YEAR_NUMBER;
M : MONTH_NUMBER;
D : DAY_NUMBER;
S : DAY_DURATION;

begin

-- Use CALENDAR functions to get year, month, day of ABT
Y := YEAR_NUMBER ( ABT.ABS_YEAR );
M := MONTH_NUMBER ( ABT.ABS_MONTH );
D := DAY_NUMBER ( ABT.ABS_DAY );

-- Convert hours, minutes, seconds of ABT to seconds ( RELATIVE_TIME )
RTM := MAKE_RELATIVE_TIME ( ABT.ABS_HOUR,
        ABT.ABS_MINUTE,
        ABT.ABS_SECONDS );

-- Convert RELATIVE_TIME type of RTM to DAY_DURATION subtype,
-- then represent all values in terms of CALENDAR.TIME
S := DAY_DURATION ( RTM );
TM := TIME_OF ( Y, M, D, S );

-- Use CALENDAR "+" function to add input objects
TM := CALENDAR."+" ( TM, DURATION ( RT ) );

-- Extract necessary values to fill ABSOLUTE_TIME returned variable
Y := YEAR ( TM );
M := MONTH ( TM );
D := DAY ( TM );
S := SECONDS ( TM );

-- Fill ABSOLUTE_TIME returned variable
RABT.ABS_YEAR := NATURAL ( Y );
RABT.ABS_MONTH := NATURAL ( M );
RABT.ABS_DAY := NATURAL ( D );
RABT.ABS_HOUR := HOURS ( RELATIVE_TIME ( S ) );
RABT.ABS_MINUTE := MINUTES ( RELATIVE_TIME ( S ) );
RABT.ABS_SECONDS := SECONDS ( RELATIVE_TIME ( S ) );

return RABT;
```

```
end "+";

--
.................................."+"...............................
function "+"
( RT : RELATIVE_TIME;
ABT : ABSOLUTE_TIME ) return ABSOLUTE_TIME is

RABT : ABSOLUTE_TIME;
RTM : RELATIVE_TIME;
TM : TIME;
Y : YEAR_NUMBER;
M : MONTH_NUMBER;
D : DAY_NUMBER;
S : DAY_DURATION;

begin

-- Use CALENDAR functions to get year, month, day of ABT
Y := YEAR_NUMBER ( ABT.ABS_YEAR );
M := MONTH_NUMBER ( ABT.ABS_MONTH );
D := DAY_NUMBER ( ABT.ABS_DAY );

-- Convert hours, minutes, seconds of ABT to seconds ( RELATIVE_TIME )
RTM := MAKE_RELATIVE_TIME ( ABT.ABS_HOUR,
ABT.ABS_MINUTE,
        ABT.ABS_SECONDS );

-- Convert RELATIVE_TIME type of RTM to DAY_DURATION subtype,
-- then represent all values in terms of CALENDAR.TIME
S := DAY_DURATION ( RTM );
TM := TIME_OF ( Y, M, D, S );

-- Use CALENDAR "+" function to add input objects
TM := CALENDAR."+" ( TM, DURATION ( RT ) );

-- Extract necessary values to fill ABSOLUTE_TIME returned variable
Y := YEAR ( TM );
M := MONTH ( TM );
D := DAY ( TM );
```

```
S := SECONDS ( TM );

-- Fill ABSOLUTE_TIME returned variable
RABT.ABS_YEAR := NATURAL ( Y );
RABT.ABS_MONTH := NATURAL ( M );
RABT.ABS_DAY := NATURAL ( D );
RABT.ABS_HOUR := HOURS ( RELATIVE_TIME ( S ) );
RABT.ABS_MINUTE := MINUTES ( RELATIVE_TIME ( S ) );
RABT.ABS_SECONDS := SECONDS ( RELATIVE_TIME ( S ) );

return RABT;

end "+";

--...................................."-
"....................................
function "-"
( T1, T2 : ABSOLUTE_TIME ) return RELATIVE_TIME is

TM1,
TM2 : TIME;
DUR : DURATION;
Y1,
Y2 : YEAR_NUMBER;
M1,
M2 : MONTH_NUMBER;
D1,
D2 : DAY_NUMBER;
S1,
S2 : DAY_DURATION;
RT1,
RT2 : RELATIVE_TIME;

begin

-- Use CALENDAR functions to get year, month, day of T1, T2
Y1 := YEAR_NUMBER ( T1.ABS_YEAR );
Y2 := YEAR_NUMBER ( T2.ABS_YEAR );
M1 := MONTH_NUMBER ( T1.ABS_MONTH );
M2 := MONTH_NUMBER ( T2.ABS_MONTH );
D1 := DAY_NUMBER ( T1.ABS_DAY );
```

```
 D2 := DAY_NUMBER ( T2.ABS_DAY );

 -- Convert hours, minutes, seconds of T1, T2 to seconds ( RELATIVE_TIME
)
 RT1 := MAKE_RELATIVE_TIME ( T1.ABS_HOUR,
         T1.ABS_MINUTE,
         T1.ABS_SECONDS );
 RT2 := MAKE_RELATIVE_TIME ( T2.ABS_HOUR,
         T2.ABS_MINUTE,
         T2.ABS_SECONDS );

 -- Convert RELATIVE_TIME types of T1, T2 to DAY_DURATION subtype,
 -- then represent all values in terms of CALENDAR.TIME
 S1 := DAY_DURATION ( RT1 );
 S2 := DAY_DURATION ( RT2 );
 TM1 := TIME_OF ( Y1, M1, D1, S1 );
 TM2 := TIME_OF ( Y2, M2, D2, S2 );

 -- Use CALENDAR "-" function to subtract T2 equivalent from T1
equivalent
 DUR := CALENDAR."-" ( TM1, TM2 );

 return RELATIVE_TIME ( DUR );

 end "-";

--
.......................................”<“.............................
 function "<"
 ( T1, T2 : ABSOLUTE_TIME ) return BOOLEAN is

 TM1,
 TM2 : TIME;
 DUR : DURATION;
 Y1,
 Y2 : YEAR_NUMBER;
 M1,
 M2 : MONTH_NUMBER;
 D1,
 D2 : DAY_NUMBER;
 S1,
 S2 : DAY_DURATION;
```

```
 RT1,
 RT2 : RELATIVE_TIME;

 begin

 -- Use CALENDAR functions to get year, month, day of T1, T2
 Y1 := YEAR_NUMBER ( T1.ABS_YEAR );
 Y2 := YEAR_NUMBER ( T2.ABS_YEAR );
 M1 := MONTH_NUMBER ( T1.ABS_MONTH );
 M2 := MONTH_NUMBER ( T2.ABS_MONTH );
 D1 := DAY_NUMBER ( T1.ABS_DAY );
 D2 := DAY_NUMBER ( T2.ABS_DAY );

 -- Convert hours, minutes, seconds of T1, T2 to seconds ( RELATIVE_TIME
)
 RT1 := MAKE_RELATIVE_TIME ( T1.ABS_HOUR,
         T1.ABS_MINUTE,
         T1.ABS_SECONDS );
 RT2 := MAKE_RELATIVE_TIME ( T2.ABS_HOUR,
         T2.ABS_MINUTE,
         T2.ABS_SECONDS );

 -- Convert RELATIVE_TIME types of T1, T2 to DAY_DURATION subtype,
 -- then represent all values in terms of CALENDAR.TIME
 S1 := DAY_DURATION ( RT1 );
 S2 := DAY_DURATION ( RT2 );
 TM1 := TIME_OF ( Y1, M1, D1, S1 );
 TM2 := TIME_OF ( Y2, M2, D2, S2 );

 -- Use CALENDAR "<" function to compare T2 equivalent to T1 equivalent
 return CALENDAR."<" ( TM1, TM2 );

 end "<";

--
.......................................................................

end ABSOLUTE_TIME_PKG;
```

APPENDIX L

DISTANCE PACKAGE

```
--
================================================================================
--
-- Authors : Richard T. Irwin
--  Willie K. Bolick
--
-- Date : 29 August 1991
--
--
.................................................................................
--
-- Description : Defines data subtype DISTANCE and associated functions
--
--
=================================================================================

package DISTANCE_PKG is

 subtype DISTANCE is FLOAT; -- Units : yards

 -- Larger than any observable range.
 UNLIMITED : constant DISTANCE := FLOAT'LAST;

 -- Unknown altitude.
 UNKNOWN : constant DISTANCE := - UNLIMITED;

 -- Converts nautical miles to yards
 function MAKE_NAUTICAL_MILES_DISTANCE
 ( NM : FLOAT ) return DISTANCE;

 -- Converts yards to nautical miles
 function DISTANCE_IN_NAUTICAL_MILES
 ( D : DISTANCE ) return FLOAT;

 pragma INLINE
 ( MAKE_NAUTICAL_MILES_DISTANCE,
```

```
 DISTANCE_IN_NAUTICAL_MILES );

end DISTANCE_PKG;

--
========================================================================
--
-- Authors : Richard T. Irwin
--  Willie K. Bolick
--
-- Date : 29 August 1991
--
--
========================================================================

package body DISTANCE_PKG is

 YDS_IN_NAUTICAL_MILE : constant FLOAT := 6080.2 / 3.0;

--
..........................MAKE_NAUTICAL_MILES_DISTANCE.................
 function MAKE_NAUTICAL_MILES_DISTANCE
 ( NM : FLOAT ) return DISTANCE is
 begin
 return NM * YDS_IN_NAUTICAL_MILE;
 end MAKE_NAUTICAL_MILES_DISTANCE;

--
..........................DISTANCE_IN_NAUTICAL_MILES...................
 function DISTANCE_IN_NAUTICAL_MILES
 ( D : DISTANCE ) return FLOAT is
 begin
 return D / YDS_IN_NAUTICAL_MILE;
 end DISTANCE_IN_NAUTICAL_MILES;

--......................................................................
end DISTANCE_PKG;
```

APPENDIX M

GLOBAL OBSERVATION PACKAGE

```
--
==========================================================================
--
-- Authors : Richard T. Irwin
--  Willie K. Bolick
--
-- Date : 29 August 1991
--
--
..........................................................................
--
-- Description : Defines data type GLOBAL_OBSERVATION
--
--
==========================================================================

with GLOBAL_POSITION_PKG, VELOCITY_PKG, ABSOLUTE_TIME_PKG;

use GLOBAL_POSITION_PKG, VELOCITY_PKG, ABSOLUTE_TIME_PKG;

package GLOBAL_OBSERVATION_PKG is

 type GLOBAL_OBSERVATION is
 record
 POSITION : GLOBAL_POSITION;
 COURSE_AND_SPEED : VELOCITY;
 OBSERVATION_TIME : ABSOLUTE_TIME;
 end record;

end GLOBAL_OBSERVATION_PKG;
```

APPENDIX N

GLOBAL POSITION PACKAGE

```
--
==========================================================================
--
-- Authors : Richard T. Irwin
--  Willie K. Bolick
--
-- Date : 29 August 1991
--
--
..........................................................................
--
-- Description : Defines abstract data type GLOBAL_POSITION and
associated
--  functions/procedures
--
--
==========================================================================

with RELATIVE_POSITION_PKG, ANGLE_PKG, DISTANCE_PKG;

use RELATIVE_POSITION_PKG, ANGLE_PKG, DISTANCE_PKG;

package GLOBAL_POSITION_PKG is

 type GLOBAL_POSITION is private; -- Earth coordinates.
 type NORTH_SOUTH is ( N, S );-- Specifies latitude hemispere
 type EAST_WEST is ( E, W );-- Specifies longitude hemisphere

 -- Converts lat/long degrees, minutes, seconds to GLOBAL_POSITION
 function MAKE_GLOBAL_POSITION
 ( LATITUDE_DIRECTION : NORTH_SOUTH;
 LATITUDE_DEGREES : NATURAL;
 LATITUDE_MINUTES : NATURAL;
 LATITUDE_SECONDS : NATURAL;
 LONGITUDE_DIRECTION : EAST_WEST;
 LONGITUDE_DEGREES : NATURAL;
```

```
LONGITUDE_MINUTES : NATURAL;
LONGITUDE_SECONDS : NATURAL ) return GLOBAL_POSITION;

-- Finds bearing & range ( RELATIVE_POSITION ) from 1 earth coordinate
to
-- another
function FIND_RELATIVE_POSITION
( CONTACT,
REFERENCE_POINT : GLOBAL_POSITION ) return RELATIVE_POSITION;

-- Returns an earth coordinate, given 1 earth coordinate and a bearing &
range
-- ( RELATIVE_POSITION )
function FIND_GLOBAL_POSITION
( OFFSET : RELATIVE_POSITION;
REFERENCE_POINT : GLOBAL_POSITION ) return GLOBAL_POSITION;

-- Returns length of the great circle path from p1 to p2
function GREAT_CIRCLE_DISTANCE
( P1,
P2 : GLOBAL_POSITION ) return DISTANCE;

-- Returns true bearing at position p1 of the great circle path from p1
to p2
function GREAT_CIRCLE_BEARING
( P1,
P2 : GLOBAL_POSITION ) return ANGLE;

-- Returns latitude ( in familiar terms, degrees, minutes, seconds ) of
a
-- given GLOBAL_POSITION
procedure GET_LATITUDE
( POSITION : in GLOBAL_POSITION;
DIRECTION : out NORTH_SOUTH;
DEGREES : out NATURAL;
MINUTES : out NATURAL;
SECONDS : out NATURAL );

-- Returns longitude ( in familiar terms, degrees, minutes, seconds ) of
a
-- given GLOBAL_POSITION
procedure GET_LONGITUDE
```

```
 ( POSITION : in GLOBAL_POSITION;
 DIRECTION : out EAST_WEST;
 DEGREES : out NATURAL;
 MINUTES : out NATURAL;
 SECONDS : out NATURAL );

 pragma INLINE ( MAKE_GLOBAL_POSITION, FIND_RELATIVE_POSITION,
      FIND_GLOBAL_POSITION, GREAT_CIRCLE_DISTANCE,
      GREAT_CIRCLE_BEARING, GET_LATITUDE, GET_LONGITUDE );

private

 type GLOBAL_POSITION is
 record
 THETA : ANGLE; -- Longitude angle in radians, -2pi to 2pi
       -- 0.0 = Greenwich Meridian
 -- 0.0 to 2pi = East longitude
 PHI : AZIMUTH; -- Latitude angle in radians, -pi to pi
       -- 0.0 = equator
       -- 0.0 to pi = North latitude
 end record;

end GLOBAL_POSITION_PKG;

--
========================================================================
--
-- Authors : Richard T. Irwin
--  Willie K. Bolick
--
-- Date : 29 August 1991
--
--
========================================================================

with MATH, VECTOR_2_PKG;

use VECTOR_2_PKG;

package body GLOBAL_POSITION_PKG is
```

```
--
............................MAKE_GLOBAL_POSITION......................
 function MAKE_GLOBAL_POSITION
 ( LATITUDE_DIRECTION : NORTH_SOUTH;
 LATITUDE_DEGREES : NATURAL;
 LATITUDE_MINUTES : NATURAL;
 LATITUDE_SECONDS : NATURAL;
 LONGITUDE_DIRECTION : EAST_WEST;
 LONGITUDE_DEGREES : NATURAL;
 LONGITUDE_MINUTES : NATURAL;
 LONGITUDE_SECONDS : NATURAL ) return GLOBAL_POSITION is

 LAT_DEG,
 LONG_DEG : FLOAT;
 GP : GLOBAL_POSITION;

 begin

 -- Convert latitude, longitude to seconds
 LAT_DEG := FLOAT ( LATITUDE_DEGREES * 3600 + LATITUDE_MINUTES * 60 +
 LATITUDE_SECONDS );
 LONG_DEG := FLOAT ( LONGITUDE_DEGREES * 3600 + LONGITUDE_MINUTES * 60 +
 LONGITUDE_SECONDS );

 -- Convert longitude seconds to radians (0..PI = east, -PI..0 = west)
 GP.THETA := ANGLE ( LONG_DEG / 3600.0 * MATH.PI / 180.0 );

 if LONGITUDE_DIRECTION = W then
 GP.THETA := -GP.THETA;
 end if;

 -- Convert latitude seconds to radians (0..PI/2 = north, -PI/2..0 =
south)
 GP.PHI := AZIMUTH ( LAT_DEG / 3600.0 * MATH.PI / 180.0 );

 if LATITUDE_DIRECTION = S then
 GP.PHI := -GP.PHI;
 end if;

 return GP;
```

```
end MAKE_GLOBAL_POSITION;

--
.............................FIND_RELATIVE_POSITION...................
function FIND_RELATIVE_POSITION
( CONTACT,
REFERENCE_POINT : GLOBAL_POSITION ) return RELATIVE_POSITION is

DELTA_LAT_IN_NM,
DELTA_LONG_IN_NM : FLOAT; -- In nautical miles
DELTA_LAT_IN_RADIANS : AZIMUTH; -- In radians
DELTA_LONG_IN_RADIANS : ANGLE; -- In radians
CTC_REL_POS : RELATIVE_POSITION;

begin

-- Compute change in latitude ( radians )
DELTA_LAT_IN_RADIANS := CONTACT.PHI - REFERENCE_POINT.PHI;

--If E / W hemisphere change over International Date Line
if ( CONTACT.THETA * REFERENCE_POINT.THETA < 0.0 ) and
( ABS ( CONTACT.THETA - REFERENCE_POINT.THETA ) ) > MATH.PI then

-- If going East to West
if REFERENCE_POINT.THETA > 0.0 then
DELTA_LONG_IN_RADIANS := MATH.PI * 2.0 - ( REFERENCE_POINT.THETA -
              CONTACT.THETA );
-- If going West to East
else
DELTA_LONG_IN_RADIANS := - MATH.PI * 2.0 - ( REFERENCE_POINT.THETA -
              CONTACT.THETA );
end if;

-- No change in E / W hemispheres
else
DELTA_LONG_IN_RADIANS := CONTACT.THETA - REFERENCE_POINT.THETA;
end if;

-- Convert lat/long change to nautical miles
-- 1 degree ( in radians ) of change = 60 miles
DELTA_LAT_IN_NM := DELTA_LAT_IN_RADIANS * 180.0 / MATH.PI * 60.0;
```

```
DELTA_LONG_IN_NM := DELTA_LONG_IN_RADIANS * 180.0 / MATH.PI * 60.0;

-- Convert the changes in lat/long to DISTANCE ( yards )
-- then initialize the RELATIVE_POSITION (2-D vector)
CTC_REL_POS := MAKE_CARTESIAN_VECTOR_2
     ( FLOAT ( MAKE_NAUTICAL_MILES_DISTANCE
        ( DELTA_LONG_IN_NM ) ),
     FLOAT ( MAKE_NAUTICAL_MILES_DISTANCE
        ( DELTA_LAT_IN_NM ) ) );

return CTC_REL_POS;

end FIND_RELATIVE_POSITION;

--
.............................FIND_GLOBAL_POSITION.......................
function FIND_GLOBAL_POSITION
( OFFSET : RELATIVE_POSITION;
REFERENCE_POINT : GLOBAL_POSITION ) return GLOBAL_POSITION is

DELTA_LAT_IN_NM,
DELTA_LONG_IN_NM : FLOAT; -- in nautical miles
DELTA_LAT_IN_RADIANS : AZIMUTH; -- in radians
DELTA_LONG_IN_RADIANS : ANGLE; -- in radians
CTC_POSITION : GLOBAL_POSITION;

begin

-- Get changes in lat/long & convert to nautical miles
DELTA_LAT_IN_NM := DISTANCE_IN_NAUTICAL_MILES ( Y_COORDINATE ( OFFSET )
);
DELTA_LONG_IN_NM := DISTANCE_IN_NAUTICAL_MILES ( X_COORDINATE ( OFFSET
) );

-- Convert NM to radians
DELTA_LAT_IN_RADIANS := AZIMUTH ( DEGREES_TO_RADIANS
          ( DELTA_LAT_IN_NM / 60.0 ) );
DELTA_LONG_IN_RADIANS := DEGREES_TO_RADIANS ( DELTA_LONG_IN_NM / 60.0
);

-- If the target lies on the other side of the pole, don't
-- make the resultant latitude > 90 degrees
```

```
 if ABS ( REFERENCE_POINT.PHI + DELTA_LAT_IN_RADIANS ) > MATH.PI / 2.0
then

 -- If going over the south pole
 if DELTA_LAT_IN_RADIANS < 0.0 then
 CTC_POSITION.PHI := - MATH.PI - ( REFERENCE_POINT.PHI +
             DELTA_LAT_IN_RADIANS );
 -- Going over the north pole
 else
 CTC_POSITION.PHI := MATH.PI - ( REFERENCE_POINT.PHI +
            DELTA_LAT_IN_RADIANS );
 end if;

 -- If we cross the n/s pole, we also change e/w hemispheres
 DELTA_LONG_IN_RADIANS := DELTA_LONG_IN_RADIANS + MATH.PI;

 -- Not going over the n/s pole
 else
 -- Assign target's latitude ( in radians )
 CTC_POSITION.PHI := REFERENCE_POINT.PHI + DELTA_LAT_IN_RADIANS;

 end if;

 -- If target lies in other e/w hemisphere
 if ABS ( REFERENCE_POINT.THETA + DELTA_LONG_IN_RADIANS ) > MATH.PI then

 -- Target is in western hemisphere
 if DELTA_LONG_IN_RADIANS < 0.0 then
 DELTA_LONG_IN_RADIANS := DELTA_LONG_IN_RADIANS + MATH.PI * 2.0;

 -- Target is in eastern hemisphere
 else
 DELTA_LONG_IN_RADIANS := DELTA_LONG_IN_RADIANS - MATH.PI * 2.0;
 end if;

 end if;

 -- Assign target's longitude
 CTC_POSITION.THETA := REFERENCE_POINT.THETA + DELTA_LONG_IN_RADIANS;

 return CTC_POSITION;
```

```
end FIND_GLOBAL_POSITION;

--
..............................GREAT_CIRCLE_DISTANCE.....................
function GREAT_CIRCLE_DISTANCE
( P1,
P2 : GLOBAL_POSITION ) return DISTANCE is

CTC_RG_BRG : RELATIVE_POSITION;

begin

-- Find where P2 is in relation to P1 ( bearing & range )
CTC_RG_BRG := FIND_RELATIVE_POSITION ( P1, P2 );

-- Return only the range ( great circle )
return RANGE_OF ( CTC_RG_BRG );

end GREAT_CIRCLE_DISTANCE;

--............................GREAT_CIRCLE_BEARIN
function GREAT_CIRCLE_BEARING
( P1, -- From
P2 -- To
   : GLOBAL_POSITION ) return ANGLE is

CTC_RG_BRG : RELATIVE_POSITION;

begin

-- Find where P2 is in relation to P1 ( bearing & range )
CTC_RG_BRG := FIND_RELATIVE_POSITION ( P1, P2 );

-- Return only the bearing ( great circle )
return BEARING_TO ( CTC_RG_BRG );

end GREAT_CIRCLE_BEARING;
```

```
--
..............................GET_LATITUDE..............................
procedure GET_LATITUDE
( POSITION : in GLOBAL_POSITION;
DIRECTION : out NORTH_SOUTH;
DEGREES : out NATURAL;
MINUTES : out NATURAL;
SECONDS : out NATURAL ) is

PH : AZIMUTH := POSITION.PHI;
DEG : NATURAL;
MIN : NATURAL;
SEC : NATURAL;

begin

-- If the value of target's PHI < 0.0, it's in the southern hemisphere
if PH < 0.0 then
DIRECTION := S;
PH := -PH;
else
DIRECTION := N;
end if;

-- Convert latitude ( radians ) to seconds
SEC := NATURAL ( FLOAT ( PH ) * 180.0 / MATH.PI * 3600.0 );

-- Calculate degrees, minutes, seconds
DEG := SEC / 3600;
MIN := ( SEC - DEG * 3600 ) / 60;
SEC := SEC - DEG * 3600 - MIN * 60;
DEGREES := DEG;
MINUTES := MIN;
SECONDS := SEC;

end GET_LATITUDE;

--
................................GET_LONGITUDE.........................
procedure GET_LONGITUDE
( POSITION : in GLOBAL_POSITION;
```

```
DIRECTION : out EAST_WEST;
DEGREES : out NATURAL;
MINUTES : out NATURAL;
SECONDS : out NATURAL ) is

TH : ANGLE := POSITION.THETA;
DEG : NATURAL;
MIN : NATURAL;
SEC : NATURAL;

begin

-- If the value of target's THETA < 0.0, it's in the western hemisphere
if TH < 0.0 then
DIRECTION := W;
TH := -TH;
else
DIRECTION := E;
end if;

-- Convert longitude ( radians ) to seconds
SEC := NATURAL ( FLOAT ( TH ) * 180.0 / MATH.PI * 3600.0 );

-- Calculate degrees, minutes, seconds
DEG := SEC / 3600;
MIN := ( SEC - DEG * 3600 ) / 60;
SEC := SEC - DEG * 3600 - MIN * 60;
DEGREES := DEG;
MINUTES := MIN;
SECONDS := SEC;

end GET_LONGITUDE;
-- ..................................................................
end GLOBAL_POSITION_PKG;
```

APPENDIX O

RELATIVE OBSERVATION PACKAGE

```
--
==========================================================================
--
-- Authors : Richard T. Irwin
--  Willie K. Bolick
--
-- Date : 29 August 1991
--
--
..........................................................................
--
-- Description : Defines data type RELATIVE_OBSERVATION
--
--
==========================================================================

with RELATIVE_POSITION_PKG, CALENDAR;

use RELATIVE_POSITION_PKG, CALENDAR;

package RELATIVE_OBSERVATION_PKG is

 type RELATIVE_OBSERVATION is
 record
 POSITION : RELATIVE_POSITION;
 OBSERVATION_TIME : TIME;
 end record;

end RELATIVE_OBSERVATION_PKG;
```

APPENDIX P

RELATIVE POSITION PACKAGE

```
--
================================================================================
--
-- Authors : Richard T. Irwin
--  Willie K. Bolick
--
-- Date : 29 August 1991
--
................................................................................
--
-- Description : Defines data type RELATIVE_POSITION and associated
functions
--
================================================================================

with VECTOR_2_PKG, DISTANCE_PKG, ANGLE_PKG;

use VECTOR_2_PKG, DISTANCE_PKG, ANGLE_PKG;

package RELATIVE_POSITION_PKG is

 subtype RELATIVE_POSITION is VECTOR_2; -- Two dimensional position
vector.

 -- Returns the distance portion of a 2-D RELATIVE_POSITION vector
 function RANGE_OF
 ( CONTACT : RELATIVE_POSITION ) return DISTANCE
  renames VECTOR_2_PKG.LENGTH;

 -- Returns the bearing portion of a 2-D RELATIVE_POSITION vector
 function BEARING_TO
 ( CONTACT : RELATIVE_POSITION ) return ANGLE
  renames VECTOR_2_PKG.DIRECTION;

 pragma INLINE ( RANGE_OF, BEARING_TO );

end RELATIVE_POSITION_PKG;
```

APPENDIX Q

TRACK DATABASE PACKAGE

```
--
=====================================================================
-- Authors : Richard T. Irwin
--  Willie K. Bolick
--
-- Date : 29 August 1991
--
--
.....................................................................
--
-- Description : Defines abstract data type TRACK_DATABASE and
associated
--  functions and procedures
--
--
=====================================================================
with TRACK_PKG;

use TRACK_PKG;

package TRACK_DATABASE_PKG is

 type TRACK_DATABASE is private;

 -- Determines whether or not a TRACK is active
 function ACTIVE_TRACK
 ( DBASE : TRACK_DATABASE ) return BOOLEAN;

 -- Restores active TRACK to database before new one is activated
 procedure RESTORE_ALTERED_TRACK_TO_DATABASE
 ( TRAK : in TRACK;
 DBASE : in out TRACK_DATABASE );

 -- Finds a TRACK in the database by track number
 procedure FIND_TRACK_IN_DBASE
 ( TRAK_NUM : in NATURAL;
```

```
TRAK : in out TRACK;
DBASE : in out TRACK_DATABASE );

-- Adds a new TRACK to the database
procedure ADD_TRACK_TO_DBASE
( TRAK : in TRACK;
DBASE : in out TRACK_DATABASE );

-- TRACKS object & all associated observations of that TRACK
-- are purged. Only the currently active TRACK can be deleted.
procedure DROP_TRACK_FROM_DBASE
( DBASE : in out TRACK_DATABASE );

-- Drops all TRACKS from the database and sends them to history
-- Should be automatically invoked upon termination of main program
procedure PURGE_ENTIRE_DBASE
( DBASE : in out TRACK_DATABASE );

pragma INLINE ( ACTIVE_TRACK, RESTORE_ALTERED_TRACK_TO_DATABASE,
     FIND_TRACK_IN_DBASE, ADD_TRACK_TO_DBASE,
     DROP_TRACK_FROM_DBASE, PURGE_ENTIRE_DBASE );

private

type TRACK_NODE;-- Elements of the TRACK_DATABASE

type TRACK_PTR is access TRACK_NODE;

type TRACK_NODE is
record
TRAK : TRACK;
NEXT_TRACK : TRACK_PTR;
end record;

type TRACK_DATABASE is
record
OWNSHIP_POSITION : TRACK_PTR; -- Points to OWNSHIP TRACK
CURRENT_TRACK_POSITION : TRACK_PTR; -- Points to currently active TRACK
PRIOR_TRACK_POSITION : TRACK_PTR; -- Points to TRACK before active
            -- TRACK for relink purposes after
            -- active TRACK is deleted
```

```
 end record;

end TRACK_DATABASE_PKG;

--
===========================================================================
--
-- Authors : Richard T. Irwin
--  Willie K. Bolick
--
-- Date : 29 August 1991
--
--
==========================================================================

with UNCHECKED_DEALLOCATION;

package body TRACK_DATABASE_PKG is

 procedure FREE_TRK is
 new UNCHECKED_DEALLOCATION ( TRACK_NODE, TRACK_PTR );

--
.................................ACTIVE_TRACK...........................
 function ACTIVE_TRACK
 ( DBASE : TRACK_DATABASE ) return BOOLEAN is

 begin

 if DBASE.CURRENT_TRACK_POSITION = null then
 return FALSE;
 end if;

 return TRUE;

 end ACTIVE_TRACK;

--
........................RESTORE_ALTERED_TRACK_TO_DATABASE..............
 procedure RESTORE_ALTERED_TRACK_TO_DATABASE
 ( TRAK : in TRACK; -- altered TRACK
 DBASE : in out TRACK_DATABASE ) is
```

```
begin

-- If currently active TRACK was not deleted
if ACTIVE_TRACK ( DBASE ) then

-- Restore currently active TRACK
DBASE.CURRENT_TRACK_POSITION.TRAK := TRAK;

-- Restore OWNSHIP TRACK, if necessary
if DBASE.CURRENT_TRACK_POSITION = DBASE.OWNSHIP_POSITION then
DBASE.OWNSHIP_POSITION.TRAK := TRAK;
end if;

end if;

end RESTORE_ALTERED_TRACK_TO_DATABASE;

--
..............................FIND_TRACK_IN_DBASE......................
procedure FIND_TRACK_IN_DBASE
( TRAK_NUM : in NATURAL;
TRAK : in out TRACK;
DBASE : in out TRACK_DATABASE ) is

begin

-- Restore currently active TRACK before reassigning current pointer
RESTORE_ALTERED_TRACK_TO_DATABASE ( TRAK, DBASE );

if TRAK_NUM /= 0 then-- not OWNSHIP

DBASE.CURRENT_TRACK_POSITION := DBASE.OWNSHIP_POSITION.NEXT_TRACK;
DBASE.PRIOR_TRACK_POSITION := DBASE.OWNSHIP_POSITION;

while ( DBASE.CURRENT_TRACK_POSITION /= null ) and then
( TRACK_ID_NUMBER ( DBASE.CURRENT_TRACK_POSITION.TRAK ) >
  TRAK_NUM ) loop
DBASE.PRIOR_TRACK_POSITION := DBASE.CURRENT_TRACK_POSITION;
DBASE.CURRENT_TRACK_POSITION :=
DBASE.CURRENT_TRACK_POSITION.NEXT_TRACK;
```

```
end loop;

else

DBASE.CURRENT_TRACK_POSITION := DBASE.OWNSHIP_POSITION;
DBASE.PRIOR_TRACK_POSITION := null;

end if;

-- If TRACK found, return it
if ( DBASE.CURRENT_TRACK_POSITION /= null ) and then
( TRACK_ID_NUMBER ( DBASE.CURRENT_TRACK_POSITION.TRAK ) = TRAK_NUM )
then
TRAK := DBASE.CURRENT_TRACK_POSITION.TRAK;
else -- TRACK not found
DBASE.CURRENT_TRACK_POSITION := null;
end if;

end FIND_TRACK_IN_DBASE;

--
..............................ADD_TRACK_TO_DBASE.......................
procedure ADD_TRACK_TO_DBASE
( TRAK : in TRACK;
DBASE : in out TRACK_DATABASE ) is

T_P : TRACK_PTR;

begin

T_P := new TRACK_NODE;
T_P.TRAK := TRAK;

if DBASE.OWNSHIP_POSITION = null then

-- first track entered ( OWNSHIP )
DBASE.OWNSHIP_POSITION := T_P;

DBASE.PRIOR_TRACK_POSITION := T_P;

else
```

```
-- All new TRACKs are entered in the TRACK_DATABASE linked list
-- immediately following OWNSHIP
T_P.NEXT_TRACK := DBASE.OWNSHIP_POSITION.NEXT_TRACK;
DBASE.OWNSHIP_POSITION.NEXT_TRACK := T_P;
DBASE.PRIOR_TRACK_POSITION := DBASE.OWNSHIP_POSITION;

end if;

DBASE.CURRENT_TRACK_POSITION := T_P;

end ADD_TRACK_TO_DBASE;

--
...........................DROP_TRACK_FROM_DBASE.......................
procedure DROP_TRACK_FROM_DBASE
( DBASE : in out TRACK_DATABASE ) is

TR : TRACK := DBASE.CURRENT_TRACK_POSITION.TRAK;

begin

-- OWNSHIP cannot be dropped
if DBASE.CURRENT_TRACK_POSITION /= DBASE.OWNSHIP_POSITION then

-- Send TRACK data & all its observations to archive file
DELETE_TRACK_AND_SEND_TO_HISTORY ( TR );

DBASE.PRIOR_TRACK_POSITION.NEXT_TRACK :=
DBASE.CURRENT_TRACK_POSITION.NEXT_TRACK;

-- Free deleted TRACK's memory space
FREE_TRK ( DBASE.CURRENT_TRACK_POSITION );

end if;

end DROP_TRACK_FROM_DBASE;
--
..............................PURGE_ENTIRE_DBASE.......................
procedure PURGE_ENTIRE_DBASE
( DBASE : in out TRACK_DATABASE ) is
```

```
 OP : TRACK_PTR := DBASE.OWNSHIP_POSITION;
 CP, PP : TRACK_PTR;
 TRAK : TRACK := OP.TRAK;

 begin

 -- Send OWNSHIP data & all its observations to archive file
 DELETE_TRACK_AND_SEND_TO_HISTORY ( TRAK );

 -- Get next TRACK in database
 CP := OP.NEXT_TRACK;

 -- Delete TRACKs, send data to archives, and free up memory for all
 -- TRACKs in the database
 while CP /= null loop
 TRAK := CP.TRAK;
 DELETE_TRACK_AND_SEND_TO_HISTORY ( TRAK );
 PP := CP.NEXT_TRACK;
 FREE_TRK ( CP );
 CP := PP;
 end loop;

 FREE_TRK ( OP );

 DBASE.OWNSHIP_POSITION := null;
 DBASE.CURRENT_TRACK_POSITION := null;
 DBASE.PRIOR_TRACK_POSITION := null;

 end PURGE_ENTIRE_DBASE;
--
.......................................................................
end TRACK_DATABASE_PKG;
```

APPENDIX R

LINK PACKAGE

```
--
==========================================================================
--
-- Authors : Richard T. Irwin
--  Willie K. Bolick
--
-- Date : 29 August 1991
--
--
..........................................................................
--
-- Description : Defines abstract data type LINK_TYPE and associated
--  functions and procedures
--
--
==========================================================================

with TRACK_PKG, GLOBAL_POSITION_PKG, INTEGRATION_SYSTEM_PKG,
 RELATIVE_TIME_PKG, M_SERIES_MSG_PKG;

use TRACK_PKG, GLOBAL_POSITION_PKG, INTEGRATION_SYSTEM_PKG,
 RELATIVE_TIME_PKG, M_SERIES_MSG_PKG;

package LINK_PKG is

 TIME_OUT_DURATION : constant RELATIVE_TIME := 3600.0; --<<<<-----+
 -- LINK_TRACK times out after 1 hour of no updates |
 -- Actual value may differ once implemented --------------------+

 type LINK_TYPE is private;

 type LINK_TABLE;
 type LINK_PTR is access LINK_TABLE;
 type LINK_TABLE is
 record
 LINK_NUM : NATURAL; -- link assigned
```

```
TRK_NUM : NATURAL := 0; -- system assigned
CTL : CONTROL_TYPE; -- LINK, LOCAL
NEXT_LT : LINK_PTR;
end record;

-- Extracts & formats a link M series message to a LINK_TYPE that
-- is later transformed into a TRACK
function CONVERT_M_SERIES_MSG_TO_LINK_TYPE
( MSG : M_SERIES_MSG ) return LINK_TYPE;

-- Creates a TRACK under LINK control from a LINK_TYPE
procedure CREATE_LINK_TRACK
( LT : in LINK_TYPE;
L_TBL : in out LINK_PTR;
TRK : in out TRACK );

-- Adds a new observation to an existing LINK TRACK
procedure ADD_LINK_OBSERVATION
( LT : in LINK_TYPE;
TRK : in out TRACK );

-- All tracks reported over link are relative to DLRP
-- ( Data Link Reference Point )
procedure MAKE_DLRP_TRACK
( DLRP : in GLOBAL_POSITION;
TRK : in out TRACK );

procedure FIND_LINK_TYPE_IN_TABLE_BY_LINK_NUM
( LN : in NATURAL;-- link table number
LP : in LINK_PTR;
LT : out LINK_TYPE );

procedure FIND_LINK_TYPE_IN_TABLE_BY_TRK_NUM
( TN : in NATURAL;-- system assigned track number
LP : in LINK_PTR;
LT : out LINK_TYPE );

-- Updates LINK_TABLE to reflect LOCAL control so no more link
-- updates to that track will occur
procedure CHANGE_LINK_TRACK_TO_LOCAL_TRACK
( TN : in NATURAL );
```

```
-- Visits each node in LINK_TABLE
-- Calls TIME_OUT to see if outside acceptable update time
-- If no update within specified period, assume link has dropped it &
-- call DROP_LINK_TRACK_AFTER_TIME_OUT
procedure SCAN_LINK_TABLE_FOR_TIME_OUTS
( LP : in out LINK_PTR );

-- Deletes LINK TABLE entry after timeout
-- Makes call to INTEGRATION_SYSTEM to drop TRACK, if not under LOCAL
control
procedure DROP_LINK_TRACK_AFTER_TIME_OUT
( LP : in out LINK_PTR;
TRK_NUM : out NATURAL );

-- Checks LINK_TABLE to see if LINK_TYPE is under LOCAL control
function ASSIGNED_LOCAL_CONTROL
( LT : LINK_TYPE;
LP : LINK_PTR ) return BOOLEAN;

-- Calls FIND_LINK_TYPE_IN_TABLE_BY_LINK_NUM
-- Flags system to drop link track after no updates in pre-assigned
-- time period
function TIME_OUT
( LN : NATURAL ) return BOOLEAN;

pragma INLINE ( CONVERT_M_SERIES_MSG_TO_LINK_TYPE, CREATE_LINK_TRACK,
     ADD_LINK_OBSERVATION, MAKE_DLRP_TRACK,
     FIND_LINK_TYPE_IN_TABLE_BY_LINK_NUM,
     FIND_LINK_TYPE_IN_TABLE_BY_TRK_NUM,
     CHANGE_LINK_TRACK_TO_LOCAL_TRACK,
     SCAN_LINK_TABLE_FOR_TIME_OUTS, DROP_LINK_TRACK_AFTER_TIME_OUT,
     ASSIGNED_LOCAL_CONTROL, TIME_OUT );

private

type LINK_TYPE is
record
LINK_NUM : NATURAL;
REL_POS_FM_DLRP : RELATIVE_POSITION;
TIME_OF_OBS : ABSOLUTE_TIME;
```

```
 TRK_CAT : TRACK_CATEGORY;
 TRK_ID : IDENTITY_TYPE;
 ALTITUDE : DISTANCE := 0.0;
 end record;

end LINK_PKG;
```

APPENDIX S

SYSTEM STATUS PACKAGE

```
--
==========================================================================
--
-- Authors : Richard T. Irwin
--  Willie K. Bolick
--
-- Date : 29 August 1991
--
--
..........................................................................
--
-- Description : Defines abstract data type SYSTEM_STATUS and associated
--  functions and procedures
--
--
==========================================================================

package SYSTEM_STATUS_PKG is

 type STATUS is ( UP, DOWN );
 type SENSOR is ( LINK, GPS, RADAR, PITSWORD, GYRO, FATHOMETER );
 type SYSTEM_STATUS is private;

 -- Retrieves status of a particular sensor
 function GET_STATUS
 ( SYS_STATUS : SYSTEM_STATUS;
 SENSER : SENSOR ) return STATUS;

 -- Sets the status of a particular sensor
 procedure SET_STATUS
 ( SYS_STATUS : out SYSTEM_STATUS;
 SENSER : in SENSOR;
 UP_DOWN : in STATUS );

 pragma INLINE ( GET_STATUS, SET_STATUS );
```

```
private

 type SYSTEM_STATUS is
 record
 LINK_STATUS : STATUS := DOWN;
 GPS_STATUS : STATUS := DOWN;
 RADAR_STATUS : STATUS := DOWN;
 PITSWORD_STATUS : STATUS := DOWN;
 GYRO_STATUS : STATUS := DOWN;
 FATHOMETER_STATUS : STATUS := DOWN;
 end record;

end SYSTEM_STATUS_PKG;

--
========================================================================
--
-- Authors : Richard T. Irwin
--  Willie K. Bolick
--
-- Date : 29 August 1991
--
--
========================================================================
package body SYSTEM_STATUS_PKG is

--
................................GET_STATUS................................
 function GET_STATUS
 ( SYS_STATUS : SYSTEM_STATUS;
 SENSER : SENSOR ) return STATUS is

 begin

 case SENSER is
 when LINK =>
 return SYS_STATUS.LINK_STATUS;
 when GPS =>
 return SYS_STATUS.GPS_STATUS;
 when RADAR =>
 return SYS_STATUS.RADAR_STATUS;
 when PITSWORD =>
```

```
 return SYS_STATUS.PITSWORD_STATUS;
 when GYRO =>
 return SYS_STATUS.GYRO_STATUS;
 when FATHOMETER =>
 return SYS_STATUS.FATHOMETER_STATUS;
 end case;

 end GET_STATUS;
--
................................SET_STATUS................................
 procedure SET_STATUS
 ( SYS_STATUS : out SYSTEM_STATUS;
 SENSER : in SENSOR;
 UP_DOWN : in STATUS ) is

 begin

 case SENSER is
 when LINK =>
 SYS_STATUS.LINK_STATUS := UP_DOWN;
 when GPS =>
 SYS_STATUS.GPS_STATUS := UP_DOWN;
 when RADAR =>
 SYS_STATUS.RADAR_STATUS := UP_DOWN;
 when PITSWORD =>
 SYS_STATUS.PITSWORD_STATUS := UP_DOWN;
 when GYRO =>
 SYS_STATUS.GYRO_STATUS := UP_DOWN;
 when FATHOMETER =>
 SYS_STATUS.FATHOMETER_STATUS := UP_DOWN;
 end case;

 end SET_STATUS;
--
.........................................................................
end SYSTEM_STATUS_PKG;
```

APPENDIX T

NAVIGATION PACKAGE

```
--
==========================================================================
--
-- Authors : Richard T. Irwin
--  Willie K. Bolick
--
-- Date : 29 August 1991
--
--
..........................................................................
--
-- Description : Defines function GET_GPS_UPDATE
--
--
=========================================================================

with GLOBAL_OBSERVATION_PKG, TEXT_IO, GLOBAL_POSITION_PKG,
 ABSOLUTE_TIME_PKG, VECTOR_2_PKG;

use GLOBAL_OBSERVATION_PKG, TEXT_IO, GLOBAL_POSITION_PKG,
 ABSOLUTE_TIME_PKG, VECTOR_2_PKG;

package NAVIGATION_PKG is

 -- Returns current OWNSHIP's position from GPS
 function GET_GPS_UPDATE return GLOBAL_OBSERVATION;

 pragma INLINE ( GET_GPS_UPDATE );

end NAVIGATION_PKG;

package body NAVIGATION_PKG is

 function GET_GPS_UPDATE return GLOBAL_OBSERVATION is

 CHAR : CHARACTER;
```

```
THE_FILE : FILE_TYPE;
IN_STRING : BOOLEAN := FALSE; -- Start character '[' found,
           -- reading position data
LAT_DEG, -- Degrees of latitude
LONG_DEG,-- Degrees of longitude
LAT_MIN, -- Minutes of latitude
LONG_MIN,-- Minutes of longitude
LAT_SEC, -- Seconds of latitude
LONG_SEC : NATURAL;-- Seconds of longitude
LAT_MIN_FL,-- GPS output of latitude minutes
LONG_MIN_FL : FLOAT;-- GPS output of longitude minutes
LAT_DIR : NORTH_SOUTH;-- North/South latitude
LONG_DIR : EAST_WEST;-- East/West longitude
OWN_OBS : GLOBAL_OBSERVATION;-- Returned position after conversion

package NATURAL_INOUT is new INTEGER_IO ( NATURAL );
package FLOAT_INOUT is new FLOAT_IO ( FLOAT );
package N_S_INOUT is new ENUMERATION_IO ( NORTH_SOUTH );
package E_W_INOUT is new ENUMERATION_IO ( EAST_WEST );
use NATURAL_INOUT, FLOAT_INOUT, N_S_INOUT, E_W_INOUT;

begin

-- Open RS-232 comm port connected to GPS
OPEN ( THE_FILE, IN_FILE, NAME => "/dev/ttya" );

loop -- Until position data is fully read in

GET ( THE_FILE, CHAR ); -- Read the next character from the GPS string

if NOT IN_STRING then -- If start character not yet found

if CHAR = '[' then -- Start character found
  IN_STRING := TRUE;
end if;

else -- Start character has been found

 -- Skip over next 29 characters, irrelevant data
 for I in 1 .. 29 loop
GET ( THE_FILE, CHAR );
```

```
end loop;

 -- Get data that pertains to OWNSHIP's GLOBAL_POSITION
 GET ( THE_FILE, LAT_DEG, 2 );
 GET ( THE_FILE, CHAR );
 GET ( THE_FILE, LAT_MIN_FL, 7 );
 GET ( THE_FILE, LAT_DIR );
 GET ( THE_FILE, CHAR );
 GET ( THE_FILE, LONG_DEG, 3 );
 GET ( THE_FILE, CHAR );
 GET ( THE_FILE, LONG_MIN_FL, 7 );
 GET ( THE_FILE, LONG_DIR );

 -- Close the comm port
 CLOSE ( THE_FILE );

 -- GPS does not send minutes and seconds, but rather sends minutes as
 -- a floating point number. The 4 statements below convert that
 -- floating point number to minutes and seconds as required to fill a
 -- GLOBAL_POSITION.
 LAT_MIN := NATURAL ( LAT_MIN_FL - 0.5 );
 LAT_SEC := NATURAL ( ( LAT_MIN_FL - FLOAT ( LAT_MIN ) ) *
             60.0 - 0.5 );
 LONG_MIN := NATURAL ( LONG_MIN_FL - 0.5 );
 LONG_SEC := NATURAL ( ( LONG_MIN_FL - FLOAT ( LONG_MIN ) ) *
             60.0 - 0.5 );

 -- Fill the GLOBAL_OBSERVATION record with the above position,
 -- current system time, and a course and speed of 0.0, 0.0.
 -- Procedures to calculate actual course and speed are found
 -- in TRACK_PKG.
 OWN_OBS.POSITION := MAKE_GLOBAL_POSITION
  ( LAT_DIR, LAT_DEG, LAT_MIN, LAT_SEC,
      LONG_DIR, LONG_DEG, LONG_MIN, LONG_SEC );
 OWN_OBS.OBSERVATION_TIME := NOW;
 OWN_OBS.COURSE_AND_SPEED := MAKE_CARTESIAN_VECTOR_2 ( 0.0, 0.0 );

 return OWN_OBS;

end if;
```

```
 end loop;

 end GET_GPS_UPDATE;

end NAVIGATION_PKG;
```

APPENDIX U

M_SERIES_MSG_PACKAGE

```
--
==========================================================================
--
-- Authors : Richard T. Irwin
--  Willie K. Bolick
--
-- Date : 29 August 1991
--
--
..........................................................................
--
-- Description : Defines abstract data types M_SERIES_MSG,
M_SERIES_MSG_BUFFER
-- and their associated functions and procedures
--
--
==========================================================================

package M_SERIES_MSG_PKG is

 type M_SERIES_MSG is private;

 type M_SERIES_MSG_BUFFER is private;

 -- Reads in an individual M_SERIES_MSG from the LINK processor
 procedure GET_M_SERIES_MSG_FROM_LINK
 ( MSG : out M_SERIES_MSG );

 -- Loops until START TRANSMISSION signal is found on LINK.
 -- Once START TRANSMISSION signal found,
 -- calls GET_M_SERIES_MSG_FROM_LINK until END TRANSMISSION
 -- signal found.
 -- Each M_SERIES_MSG retrieved is appended to M_SERIES_MSG_BUFFER
 procedure FILL_M_SERIES_MSG_BUFFER
 ( MSG_BUFF : out M_SERIES_MSG_BUFFER );
```

```
--other functions/procedures to retrieve M_SERIES_MSG,
--M_SERIES_MSG_BUFFER record items to be completed in
--follow-on thesis work

private

  type M_SERIES_MSG is
    record
--      to be completed in follow-on thesis work
    end record;

  type M_SERIES_MSG_BUFFER is
--    some data structure of M_SERIES_MSG types

end M_SERIES_MSG_PKG;
```

APPENDIX V

PROCESS_LINK_TRACK_PACKAGE

```
--
==========================================================================
--
-- Authors : Richard T. Irwin
--  Willie K. Bolick
--
-- Date : 29 August 1991
--
--
..........................................................................
--
-- Description : Defines procedure PROCESS_MSG_BUFFER and task
-- PROCESS_LINK_TRACKS
--
--
==========================================================================

with INTEGRATION_SYSTEM_PKG, M_SERIES_MSG_PKG, LINK_PKG, TRACK_PKG,
 SYSTEM_STATUS_PKG;

use INTEGRATION_SYSTEM_PKG, M_SERIES_MSG_PKG, LINK_PKG, TRACK_PKG,
 SYSTEM_STATUS_PKG;

package PROCESS_LINK_TRACKS_PKG is

 procedure PROCESS_MSG_BUFFER
 ( MSG_BUFF : in M_SERIES_MSG_BUFFER );

 task PROCESS_LINK_TRACKS;

end PROCESS_LINK_TRACKS_PKG;



package body PROCESS_LINK_TRACKS_PKG is
```

```
procedure PROCESS_MSG_BUFFER
( MSG_BUFF : in M_SERIES_MSG_BUFFER ) is

begin

-- Uses procedures/functions in LINK_PKG, TRACK_PKG to
-- break down MSG_BUFF into individual M_SERIES_MSGs and
-- convert them to link TRACKS, altering/adding them to the
-- TRACK_DATABASE as necessary ( using INTEGRATION_SYSTEM
-- entry calls )

end PROCESS_MSG_BUFFER;

task body PROCESS_LINK_TRACKS is

MSG_BUFF : M_SERIES_MSG_BUFFER;
SENSER_STATUS : STATUS;

begin

loop

-- Get synch signal from INTEGRATION_SYSTEM_PKG.LINK_CYCLE
LINK_CYCLE.START_LINK_UPDATE;

-- See if there's anything to process
INTEGRATION_SYSTEM.GET_SENSOR_STATUS ( LINK, SENSER_STATUS );

if SENSER_STATUS = UP then

 -- Get the msg buffer
FILL_M_SERIES_MSG_BUFFER ( MSG_BUFF );

 -- Process the buffer into separate msgs and possibly LINK TRACKs
PROCESS_MSG_BUFFER ( MSG_BUFF );

end if;

end loop;

exception
```

```
 when STATUS_ERROR | CONSTRAINT_ERROR =>
 INTEGRATION_SYSTEM.SET_SENSOR_STATUS ( LINK, DOWN );

 end PROCESS_LINK_TRACKS;

end PROCESS_LINK_TRACKS_PKG;
```

APPENDIX W

RELATIVE_TIME_PACKAGE

```
--
======================================================================
--
-- Authors : Richard T. Irwin
--  Willie K. Bolick
--
-- Date : 29 August 1991
--
--
......................................................................
--
-- Description : Defines data type RELATIVE_TIME and associated
functions
--
--
======================================================================

package RELATIVE_TIME_PKG is

 subtype RELATIVE_TIME is FLOAT; -- Units : seconds

 --Returns total seconds, given hours, minutes, seconds
 function MAKE_RELATIVE_TIME
 ( HOURS, MINUTES : NATURAL;
 SECONDS : FLOAT ) return RELATIVE_TIME;

 -- Returns whole hours of a day, given seconds of a day
 function HOURS
 ( T : RELATIVE_TIME ) return NATURAL;

 -- Returns whole minutes of an hour, given seconds of a day
 function MINUTES
 ( T : RELATIVE_TIME ) return NATURAL;

 -- Returns seconds of a minute, given seconds of a day
 function SECONDS
```

```
( T : RELATIVE_TIME ) return FLOAT;

pragma INLINE ( MAKE_RELATIVE_TIME, HOURS, MINUTES, SECONDS );

end RELATIVE_TIME_PKG;

--
=======================================================================
--
-- Authors : Richard T. Irwin
--  Willie K. Bolick
--
-- Date : 29 August 1991
--
--
=======================================================================

package body RELATIVE_TIME_PKG is

--
.............................MAKE_RELATIVE_TIME........................
function MAKE_RELATIVE_TIME
( HOURS, MINUTES : NATURAL;
SECONDS : FLOAT ) return RELATIVE_TIME is

begin

return FLOAT ( HOURS * 3600 ) + FLOAT ( MINUTES * 60 ) + SECONDS;

end MAKE_RELATIVE_TIME;

--
..................................HOURS................................
function HOURS
( T : RELATIVE_TIME ) return NATURAL is

begin
return NATURAL ( T / 3600.0 - 0.5 );
end HOURS;
```

```
--
...............................MINUTES................................
 function MINUTES
 ( T : RELATIVE_TIME ) return NATURAL is

 begin
 return NATURAL ( ( T - RELATIVE_TIME ( HOURS ( T ) * 3600 ) ) /
 60.0 - 0.5 );
 end MINUTES;

--
...............................SECONDS................................
 function SECONDS
 ( T : RELATIVE_TIME ) return FLOAT is

 begin
 return T - FLOAT ( HOURS ( T ) * 3600 ) - FLOAT ( MINUTES ( T ) * 60 );
 end SECONDS;

--
......................................................................
end RELATIVE_TIME_PKG;
```

APPENDIX X

GPS CONNECTION CONSIDERATIONS

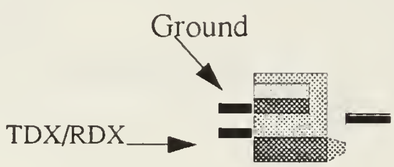
The connection between the Global_Positioning_Subsystem (Trimble 4000) and the SUN Microstation SPARCstation 2 is with a cable using the RS-232 port on the Trimble 4000 and Comm-port 1 on the SPARCstation. A proper setup of the connectors pins at each end of the cable is necessary to insure data transfer. The proper setup follows:

Trimble 4000 RS-232 connector pins (See Figure 20):

GROUND:= GROUND;

TXD(SEND):= SEND;

RXD(RECEIVE):= BLANK (no pin);



RS-232 Connector

Figure 27: GPS CONNECT

SPARCstation Comm-port 1:

GROUND:= GROUND;

TXD(SEND:= BLANK (no pin);

RXD(RECEIVE):= RECEIVE;

The network configuration in this case is simply a DTE setup in the Trimble 4000 and a DCE in the SPARCstation. This setup is necessary because the Trimble 4000 will shut down with an interrupt, if the SUN, via the RXD pin sends a "ready to receive" signal accommodating the Trimble 4000 PROTOCOL.

LIST OF REFERENCES

[Ref. 1] Commander, Naval Sea Systems Command UNCLASSIFIED Letter 9410 OPR:61Y Serial 61Y/1036 to Superintendent, Naval Postgraduate School, Subject: Statement of Work for Low Cost Combat Direction System, 20 December 1988.

[Ref. 2] Department of the Navy, (NAVSEA) 0967-LP-027-8602 System Engineering Handbook Vol I, Combat Direction System Model 5, February 1985.

[Ref. 3] Seveney, J., Steinberg, G.P., “Requirements Analysis for a Low Cost Combat Direction System”, Master’s Thesis, Naval Postgraduate School, Monterey, CA., June,1990.

[Ref. 4] Department of the Navy, Military Specification(CONFIDENTIAL NAVSEA) 0967-LP-027-8635, Combat Direction System (CDS) Specification for Surface for Surface Ships (Model 4.1) (U), Vol. 1, Revision 5, April 1988.

[Ref. 5] Department of the Navy, Military Specification(CONFIDENTIAL NAVSEA) 0967-LP-027-8635, Combat Direction System (CDS) Specification for Surface for Surface Ships (Model 4.1) (U), Vol. 2, Revision 5, April 1988.

[Ref. 6] Department of Defense Directive 5200.28, Security Requirements for Automatic Data Processing (ADP) Systems, 18 December 1972.

[Ref. 7] E. Yourdon, “Modern Structured Analysis”, Yourdon Press by Prentice Hall, N.J.,1989.

[Ref. 8] A. Tanenbaum, “Structured Computer Organization”, Prentice Hall, Inc., Englewood Cliffs, N.J.,1984.

[Ref. 9] V. Berzins and Luqi, “Software Engineering with Abstractions: An Integrated Approach to Software Development using Ada”, Addison-Wesley,1988.

[Ref. 10] D. Breen, P. Getto, A. Apodaca, "Object-Oriented Programming in a Conventional Programming Environment", Computer Society Press of the IEEE, Washington,DC,1989.

[Ref. 11] Grady Booch, "Software Engineering with Ada", Benjamin Cummings Publishing Company,1983.

[Ref. 12] R. Elmasri, S.B. Navathe, "Fundamentals of the Database Systems", The Benjamin/Cummings Publishing Company,Inc., Redwood City, CA.,1989.

[Ref. 13] S. Faulk and D. Parnas, "On Synchronization in Hard-Real_Time Systems", Comm. of the ACM 31, 3, Mar 1988, pp 274-287.

[Ref. 14] W. Lorensen, "Object-Oriented Software Development in a Non-Object-Oriented Environment," General Electric Technical Information Series Report 86CRD133,1986.

[Ref. 15] B. Meyer, "Object-Oriented Software Construction", Prentice Hall, Inc., Englewood Cliffs, NJ,1988.

[Ref. 16] Buhn, R., Karan, G., Hayse, C., Woodside, C., "Software CAD: a Revolutionary Approach", IEEE Transactions on Software Engineering, Vol 15, No.3, pp235-249, Mar 89.

[Ref. 17] Collins, M. J. Stratford-, "Ada: A Programmer's Conversion Course", R. J. Acford Press, Chichester, West Sussex, England, 1982.

[Ref. 18] Dillon, L., "Verifying General Safety Properties of Ada Tasking Programs", IEEE Transactions of Software Engineering, Vol 16, No. 1, p51-67, Jan 90.

[Ref. 19] Guaspari, D., Marceau, C., Polak, W., "Formal Verification of Ada Programs", IEEE Transactions on Software Engineering, Vol 16, No. 9, p1058-1076, Sep 90.

[Ref. 20] Jalote, Pankaj, "Functional Refinement and Nested Objects for Object-Oriented Design", IEEE Transactions on Software Engineering, Vol 15, No. 3, p264-270, Mar 89.

[Ref. 21] Moser, Louise, "Data Dependency Graphs for Ada Programs", IEEE Transactions on Software Engineering, Vol. 16, No. 5, P498-527, May 90.

[Ref. 22] Sha, L., Goodenough, J.B., "Real-Time Scheduling Theory and Ada", Computer, Vol 23, No. 4, Published by IEEE Computer Society, p53-62, Apr 90.

[Ref. 23] Sommerville, I., Welland, R., Beer, S., "Describing Software Design Methodologies", The Computer Journal, The British Computer Society, Vol 30, No. 2, p128-133, Apr 87.

[Ref. 24] Schweiger, Jeffrey M. , "Structuring a Software Tool for Detecting Deadlock Potential from the Formal Specification of a Distributed System", Master's Thesis, Naval Postgraduate School, Monterey, CA., unfinished.

[Ref. 25] Vick, C. R., Ramamoorthy, C. V., "The Handbook of Software Engineering", Van Nostrand Reinhold Company, New York, NY,1984.

[Ref. 26] Department of Defense Military Standard 2167-A, Defense System Software Development, 29 February 1988,

[Ref. 27] Department of Defense Military Standard 2168, Defense System Software Quality Program, 29 February 1988,

[Ref. 28] American National Standard Institute Military Standard 1815A-1983, Reference Manual for the Ada Programming Language, 17 February 1983.

[Ref. 29] Department of Defense, Military Standard 480, Configuration Control Engineering changes, Deviations, and waivers.

[Ref. 30] OP_SPEC 411.2 (CONFIDENTIAL), Naval Tactical Data System Model 4 Link 11 Operational Specification, Rev 2, 15 August 1985.

[Ref. 31] Department of Defense Military Standard 490, Specification Practices, May 1972.

[Ref. 32] SUN MICROSYSTEMS, SUN Systems overview, February 1986.

[Ref. 33] Skansholm, Jan, "Ada From The Beginning", Addison-Wesley publishing company, New York, NY, 1988.

[Ref. 34] Chin, Yu-Chi, "The Navigation Data Logger for a Suitcase Navigation System", Master's Thesis, Naval Postgraduate School, Monterey, CA., June,1991.

[Ref. 35] DePaula Everton G., "A Tactical Database for the Low Cost Combat Direction System", Master's Thesis, Naval Postgraduate School, Monterey, CA., December, 1990

[Ref. 36] Department of Defense Military Standard 1679, Weapons System Software Development, December 1978.

[Ref. 37] Naval Sea System Instruction 5400.57, Technical Responsibility and Authority to perform Engineering Functions for Combat Subsystems and Equipment, June 1978.

[Ref. 38] OPNAV Instruction 7700.1, Configuration of CDS and Combat Systems for General Purpose Forces, 31 August 1973.

[Ref. 39] Interview between Mr. Dan Edwards, Naval Ocean Systems Center (code 412), San Diego, CA., and the authors, 9 July 1991.

[Ref. 40] Interview between Mr. Roy, Director Research (7162), Logicon, San Diego, CA., and the authors, 5 August 1991.

[Ref. 41] Interview between Mr. George Sadowski, Fleet Combat Direction System Support Activity, Dam Neck, VA., and the authors, 22 June 1991.

INITIAL DISTRIBUTION LIST

Defense Technical Information Center 2
Cameron Station
Alexandria, VA 22304-6145

Dudley Knox Library 2
Code 52
Naval Postgraduate School
Monterey, CA 93943

Chairman, Code CS 1
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

Office of Naval Research 1
800 N. Quincy Street
Arlington, VA 22217-5000

Office of the Chief of Naval Operations 1
Code OP-941
Washington, D.C. 20350-2000

Commander 1
Naval Sea Systems Command
ATTN: LCDR Scott Kelly
Code 06D3131
Washington, D.C. 20362-5101

Commander 1
Naval Ocean Systems Center
Code 451
San Diego, CA 92152-5000

Commander 1
Naval Ocean Systems Center
Code 431
ATTN: Dan Edwards
San Diego, CA 92152-5000

Commander 1
Naval Sea Systems Command
ATTN: William L. Wilder
PMS-4123H
Arlington, VA 22202-5101

Dr. Valdis Berzins 1
Code 52Be
Computer Science Department
Naval Postgraduate School
Monterey, CA 93940

Dr. Luqi 1
Code 52Lq
Computer Science Department
Naval Postgraduate School
Monterey, CA 93940

LCDR (sel) Willie K. Bolick 1
1508 Willowbend Drive
Gautier, MS 39553

LT Richard T. Irwin 1
5130 Navajo Trail
Harrison, MI 48625

Thesis
B652 Bolick
c.1 The integration system
for the Low Cost Combat
Direction System.